BDS Exam Preparations

Challenges of Big Data Processing:

- Volume → More memory and I/O bandwidth
- Velocity → Fast Processing
- Variety → More computing paradigms

Parallel Programming

Challenges:

- Parallelism identification
- Load balancing
- Proper synchronization

Amdahl's Law

If P is the proportion of a program that can be made parallel, the limit of speedup of whole program is bounded at **Speed-up = 1/(1-P)**

Gustafson's Law

Amdahl's law assumes fixed problem size

If problem size can increase as the compute power increases, the speedup that can be achieved is much larger.

S = a(n) + p(1-a(n)))

n = problem size
p = number of processes
a(n) = serial portion at problem size n

So S → p when a(n) = 0

⇒ Larger machines are still useful
⇒ Many small chips can deliver good performance when the problem size is big

Problem: Some applications don't have bigger data set

Real world is mix of Amdahl's law and Gustafson's law

Posix Threads (Pthreads) Interface

•

Pthreads:
Standard interface for ~60 functions that
manipulate threads from C programs
–Creating and reaping threads
•`pthread_create()`
•`pthread_join()`
–Determining your thread ID
•`pthread_self()`
–Terminating threads
•`pthread_cancel()`
•`pthread_exit()`
•`exit()`[terminates all threads] ,`RET`[terminates current thread]
–Synchronizing access to shared variables
•`pthread_mutex_init`
•`pthread_mutex_[un]lock`
•`pthread_cond_init`
•`pthread_cond_[timed]wait`

Semaphores:
Non-negative global integer synchronization variables

Manipulated by *P* and *V* operations:
- *P(s):* [**`while (s == 0) wait(); s--;`**]
  - Dutch for "Proberen" (test)
- *V(s):* [**`s++;`**]
  - Dutch for "Verhogen" (increment)

OS kernel guarantees that operations between brackets [ ] are executed indivisibly

- Only one P or V operation at a time can modify s.
- When while loop in P terminates, only that P can decrement s

```
void *sum_thread(void *vargp)
{
        long sum;
        int myid = (long) vargp;
        int start = myid * sample_num / 4;
        int end = (myid+1) * sample_num / 4 - 1;

        sum = 0;
        for (int i=start; i<=end; i++)
            sum += x[i];

        sem_wait(&num_mutex);
        global_sum += sum;
        sem_post(&num_mutex);

        return NULL;
}
```

Issues with Pthreads:

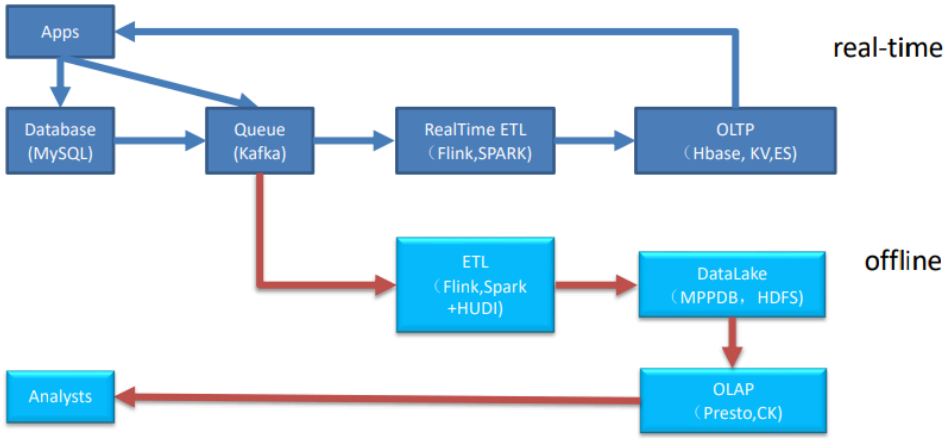
Complex for data processing

- Pthread is not always portable(e.g. Windows)
- Code to calculate addresses and loop bounds
- May not be easy to specify scheduling schemes

```
void *sum_thread(void *);

volatile long global_sum = 0; /* global */


main()
{
…
  for (int i = 0; i < N; i++)
       pthread_create(&tid[i], NULL, sum_thread, (void*)i);

   for (int i = 0; i < N; i++)
       pthread_join(tid[i], NULL);
}
```

```
void *sum_thread(void *);

volatile long global_sum = 0; /* global */


main()
{
…
  for (int i = 0; i < N; i++)
       pthread_create(&tid[i], NULL, sum_thread, (void*)i);

   for (int i = 0; i < N; i++)
       pthread_join(tid[i], NULL);
}
```

Realistic Data Pipeline



Estimating Power Efficiency of Dual Core CPU

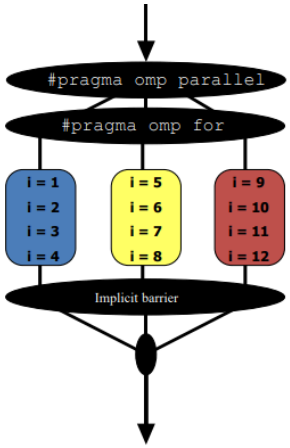
Single Core: 1 solution/s
$P = cf_1^2$
$P/2 = cf_2^2$
$F_2 = f_1/(2)^{½}$
Dual Core Throughput = $2*f2 = 2^{1/2} * f1 = 2^{1/2}$ solutions/s
⇒ With perfect parallelism

OpenMP

“Standard” API for defining multi-threaded sharedmemory programs
– Allow a programmer to **separate a program into serial regions and parallel regions**, rather than T concurrently- executing threads.
– **Hide stack management**
– Provide synchronization constructs
⇒ **ONLY SINGLE MACHINES**

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

```
#pragma omp parallel for
for(i = 1, i < 13, i++)
      c[i] = a[i] + b[i]
```



Data Sharing:

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- pthreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private
- OpenMP:
  - **shared** variables are shared
  - **private** variables are private
  - Rules for default

```
int bigdata[1024];

void* foo(void* bar) {
  int tid;

  #pragma omp parallel \
   shared ( bigdata ) \
   private ( tid )
  {
    /* Calc. here */
  }
}
```

```
int i = 0;
int n = 10;
int a = 7;

#pragma omp parallel for
for (i = 0; i < n; i++)
{
    int b = a + i;
    ...
}
```

n and a are shared. i and b are private.

Firstprivate

```
#include <omp.h>
#include <stdio.h>

int main()
{
    int sum = 0;

    #pragma omp parallel for firstprivate(sum)
    for (int i = 1; i <= 10; i ++)
    {
        sum += i;
        printf("In iteration %d, sum is %d!\n", i, sum);
    }

    printf("After the loop, sum is %d\n", sum);
}
```

- Initialize private variables from the master threads

lastprivate

```
#include <omp.h>
#include <stdio.h>

int main()
{
    int sum = 0;

    #pragma omp parallel for firstprivate(sum) lastprivate(sum)
    for (int i = 1; i <= 10; i ++)
    {
        sum += i;
        printf("In iteration %d, sum is %d!\n", i, sum);
    }

    printf("After the loop, sum is %d\n", sum);
}
```

- Pass the value of a private from the last iteration to global variable

OpenMP Critical Construct

- `#pragma omp critical [(lock_name)]`
- Defines a critical region on a structured block

Threads wait their turn –at a time, only one calls `consum()` thereby protecting RES from race conditions

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for private(B)
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical
    {
    consum (B, RES);
    }
  }
}
```

Sum of an Array

```
#include <omp.h>
#include <stdio.h>

int main()
{
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= 10; i ++)
    {
        sum += i;
        printf("In iteration %d, sum is %d!\n", i, sum);
    }

    printf("After the loop, sum is %d\n", sum);
}
```

- The schedule clause affects how loop iterations are mapped onto threads

`schedule(static [,chunk])`
- Blocks of iterations of size "chunk" to threads
- Round robin distribution
- Default=N/t

`schedule(dynamic[,chunk])`
- Threads grab "chunk" iterations
- When done with iterations, thread requests next set
- Default=1

`schedule(guided[,chunk])`
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"
- Default=1

`schedule(runtime)`
- OMP_SCHEDULE

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

MPI Programs

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

MPI Basic (Blocking) Send/Receive

MPI_SEND (start, count, datatype, dest, tag, comm)
MPI_RECV(start, count, datatype, source, tag, comm, status)

```
Process P:
send (request1,32, Q)

Process R:
send (request2, 32, Q)

Process Q:
while (true) {
    recv (received_request, 32, Any_Process);
    process received_request;
    }
```

Tags can also serves as "message type", although we should not confuse it with MPI data types

```
Process P:
send(request1, 32, Q, tag1)

Process R:
send(request2, 32, Q, tag2)

Process Q:
while (true){
    recv(received_request, 32, Any_Process, Any_Tag, Status);
    if (Status.MPI_TAG==tag1) process received_request in one way;
    if (Status.MPI_TAG==tag2) process received_request in another way;
    }
```

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

tag=1234;source=0;destination=1;count=1;

 if(myid == source){
   buffer=5678;
   MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
   printf("processor %d  sent %d\n",myid,buffer);
}
if(myid == destination){
    MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    printf("processor %d  got %d\n",myid,buffer);
}

MPI_Finalize();
```

Why Datatypes? ⇒ Support communication between processes on machines with different memory representations

Unsave message passing ⇒ Depending on the availability of system buffers:

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

Better:

- Order the operations more carefully:

Process 0	Process 1
Send(1)	Recv(0)
Recv(1)	Send(0)

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	Isend(0)
Irecv(1)	Irecv(0)
Waitall	Waitall

```
if (rank == 0) {
            MPI_Isend(..., 1, tag, MPI_COMM_WORLD, &req);
            MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
            MPI_Wait(&req, &status);
 } else if (rank == 1) {
            MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
            MPI_Send(..., 0, tag, MPI_COMM_WORLD);
}
```

Just six functions:

– MPI_INIT – MPI_FINALIZE – MPI_COMM_SIZE – MPI_COMM_RANK – MPI_SEND – MPI_RECV

Collective Operations in MPI:

- Collective operations are called by all processes in a communicator.
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI_REDUCE** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_
  MPI_
  while
    if
      I
      s
    }
    MPI
    if
                h   = 1.0 / (double) n;
                sum = 0.0;
                for (i = myid + 1; i <= n; i += numprocs) {
                  x = h * ((double)i - 0.5);
                  sum += 4.0 / (1.0 + x*x);
                }
                mypi = h * sum;
                MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                           MPI_COMM_WORLD);
                if (myid == 0)
                  printf("pi is approximately %.16f, Error is %.16f\n",
                         pi, fabs(pi - PI25DT));
              }
              MPI_Finalize();
              return 0;
            }
```

Problem with MPI: **Availability**

What if a process crashes?

→ If app runs 20 days or longer

→ Restart doesn't help

Checkpointing is required!

What are the obstacles of supporting heterogeneous node and fault tolerance

– Knowing N at the beginning

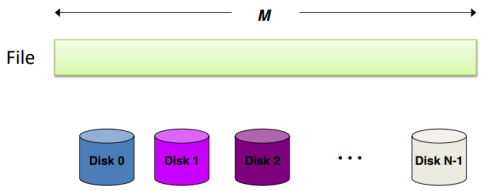
– Static partition of the workload

Parallel I/O Systems

Fault Tolerance

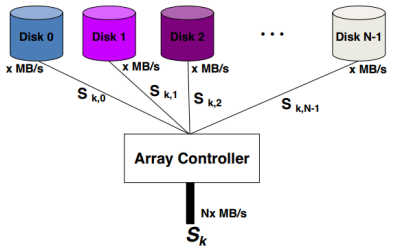
Disk Level

- Given a large file of *M* bytes file, and *N* disks
  - Find a way to distribute the file into disks.
- Objective: Read a long sequential part of the file as fast as possible

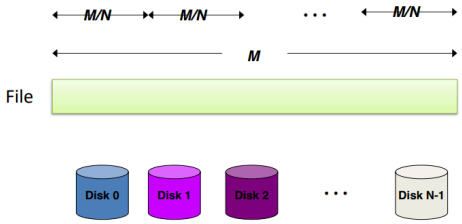


Performance Benefit

- *Sequential read or write of large file*
  - application (or I/O buffer cache) reads in multiples of S bytes
  - controller performs parallel access of N disks
  - aggregate bandwidth is N times individual disk bandwidth
    - (assumes that disk is the bottleneck)

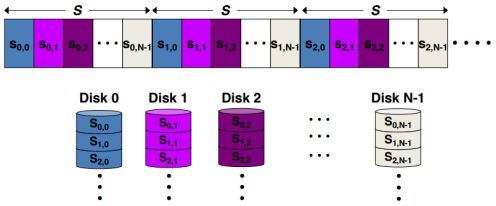


Solution 1: Chunk



Solution 2: Stripe

- *Stripe* the data across an array of disks
  - many alternative striping strategies possible
- Example: consider a big file striped across N disks
  - *stripe width* is S bytes
  - hence each *stripe unit* is S/N bytes
  - sequential read of S bytes at a time

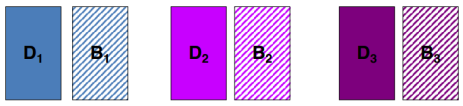


RAID

Large arrays without redundancy are too unreliable to be useful

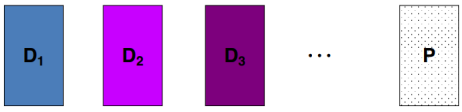
→ **"Redundant Arrays of Independent Disks"**

RAID Level 1



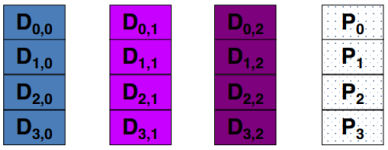
- Also known as *"mirroring"*
- To read a block:
  - read from either data disk or backup
- To write a block:
  - write both data and backup disks
- How many failure can tolerate? What is the cost for redundancy?

RAID Levels 2 & 3



- These are *bit-interleaved* schemes
- In Raid Level 2, P contains memory-style ECC
- In Raid Level 3, P contains simple parity
- Rarely used today
- Good for large data read/write, poor for small data transfer

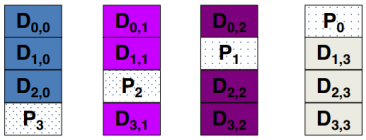
RAID Level 4



$D_{0,0} \oplus D_{0,1} \oplus D_{0,2} = P_0$

- *Block-interleaved parity*
- Wasted storage is small: one parity block for N data blocks
- Key problem:
  - parity disk becomes a hot spot
  - write access to parity disk on every write to any block

RAID Level 5



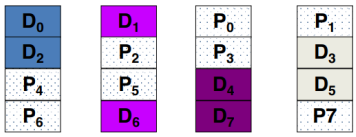
$D_{0,0} \oplus D_{0,1} \oplus D_{0,2} = P_0$

- *Rotated parity*
- Wastage is small: same as in Raid 4
- Parity update traffic is distributed across disks

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	$N/2$	$N-1$	$N-1$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	D	D	D	D
Write	D	D	$2D$	$2D$

Problem with RAID lvl 5 if failure rate not >> recovery time.

RAID Level 6



- Tolerate 2 disk failures
- Read performance is good, while write performance is a little slower
- Parity update traffic is distributed across disks

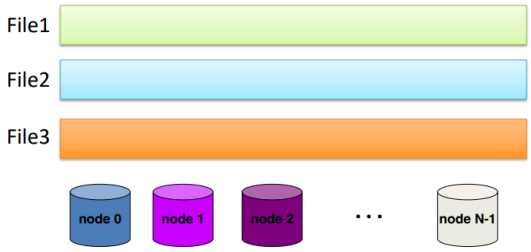
RAID provide good reliability for storage in a single node

- But it **can not tolerate node/network failure**

Node Level

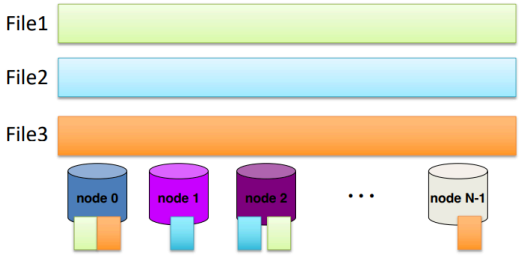
Problem Statement

- Given a set of large files and *N* machine nodes
  - Find a way to distribute the file into nodes.
- Objective:
  - Tolerate node failure
  - Read a long sequential part of the file as fast as possible
  - Write are mostly large append mode write



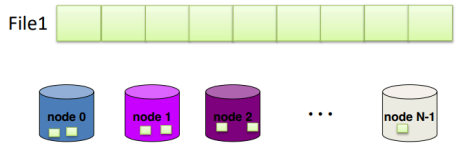
Duplicate Whole Files to Nodes

A node include all content of file.
A file is duplicated in multiple nodes. (In this example:2)
Pros and cons of this strategy?



Duplicate chunks of files

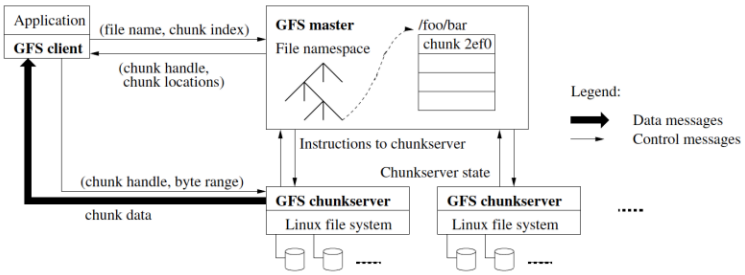
- A file is divided into fixed size chunks
- Each chunk is distributed and duplicated to a few nodes (usually 2-3)
- Pros and cons of this strategy?
  - Number of node failure tolerated?
  - Wasted space?
  - Space required to know the position of the chunk?



How to distributed 3 replicas on node/rack hierarchy?

- One replica on local node
- Second replica on a remote rack
- Third replica on same remote rack

Google File System (GFS)



Single master design
- Performance issue
  - Separate data path with control path, no data need to pass chunk server

Let's assume the system contains modest number of HUGE files, and the chunk size is large(e.g. 64MB)
- Meta data for the system is not large
- They can be put in memory of a single server, this greatly reduce the complexity of the system and could get good meta data performance.

Single master design: Simple! With Availability and performance issues

Metadata in Memory
- The entire metadata is in main memory
- No demand paging of metadata

Key GFS design decisions

- 3 replicas of data for fault tolerance
- Single master design for simplicity and performance
- Master does not in data paths
- Large chunk size for smaller meta data

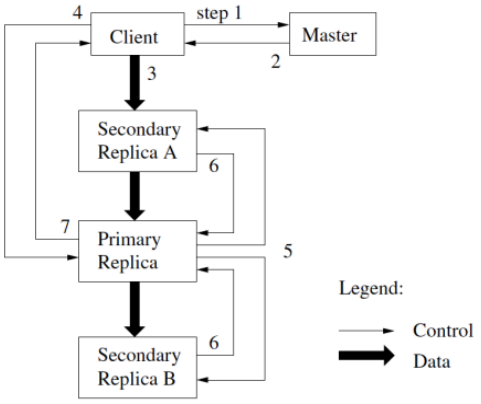
Chunk server

A Block Server
- Stores data in the local file system (e.g. ext3)
  - Each **block** of **GFS data** in a separate **file** in its local **file** system.
- Stores metadata of a block (e.g. CRC)
- Serves data and metadata to Clients

Data Correctness

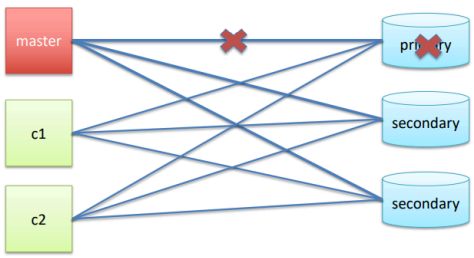
- Use Checksums to validate data
  - Use CRC32
- File Creation
  - Client computes checksum per 512 bytes
  - Chunk server stores the checksum
- File access
  - Client retrieves the data and checksum from chunk server
  - If Validation fails, Client tries other replicas

GFS Write



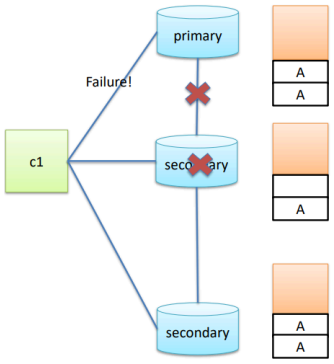
A primary is need to control the order of the concurrent write

Why we need a lease?



A lease for primary is to deal with primary failure or network failure

What happens if secondary fails



- Both secondary failure and network failure may cause write failure to client
  - Note there no undo
- Client will try the same write operations a few times(without go to master)
- If it still fails, it will restart a new write operation by talking to master
- This inconsistency between replicas are dealt with applications

GFS design make an tradeoff

- Trade ideal consistency for system simplicity
- Clients may observe
  - Reading stale data
  - Duplicate append records
- For applications such as search engine
  - This may be tolerable
- Understanding GFS design as a tradeoff among
  - Consistency,fault-tolerance,performance simplicity of design

Heartbeats

- Master sends hearbeat to the chunkservers
  - Once every 3 seconds
  - No heartbeat does not necessarily indicate chunk server failure
    - May be network failure

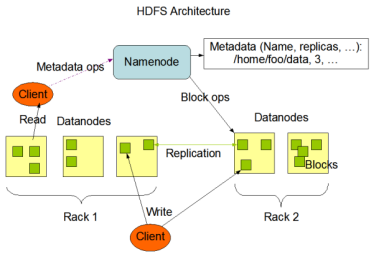
Rebalancer

- Goal: % disk full on chunk servers should be similar
  - Usually run when new chunk servers are added
  - Rebalancer is throttled to avoid network congestion

Replication Engine

- Master detects chunk server failures
  - Chooses new chunk server for new replicas
  - Balances disk usage
  - Balances communication traffic to chunk servers

HDFS Architecture



User Interface

- Commads for HDFS User:
  - hadoop dfs -mkdir /foodir
  - hadoop dfs -cat /foodir/myfile.txt
  - hadoop dfs -rm /foodir/myfile.txt
- Commands for HDFS Administrator
  - hadoop dfsadmin -report
  - hadoop dfsadmin -decommision datanodename
- Web Interface
  - http://host:port/dfshealth.jsp

Example: Count word occurrences (pseudo code)

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

MapReduce

- Doing a small piece of work, complete it and get the next piece
  - Input from the global file system, similar to the MPI_Bcast
  - Write a map() function that transfers the input piece into partial output piece
- No N is required to know at the beginning
  - Data specified by files in distributed file system
- Need a reduce support similar to MPI_Reduce
  - Write a user defined reduce() file

Example

- Page 1: the weather is good
- Page 2: today is good
- Page 3: good weather is good.

Map output

- Worker 1:
  - (the 1), (weather 1), (is 1), (good 1).
- Worker 2:
  - (today 1), (is 1), (good 1).
- Worker 3:
  - (good 1), (weather 1), (is 1), (good 1).

Reduce Input

- Worker 1:
  - (the 1)
- Worker 2:
  - (is 1), (is 1), (is 1)
- Worker 3:
  - (weather 1), (weather 1)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 1), (good 1), (good 1), (good 1)

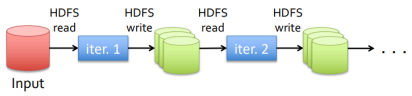
Reduce Output

- Worker 1:
  - (the 1)
- Worker 2:
  - (is 3)
- Worker 3:
  - (weather 2)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 4)

Parallelism

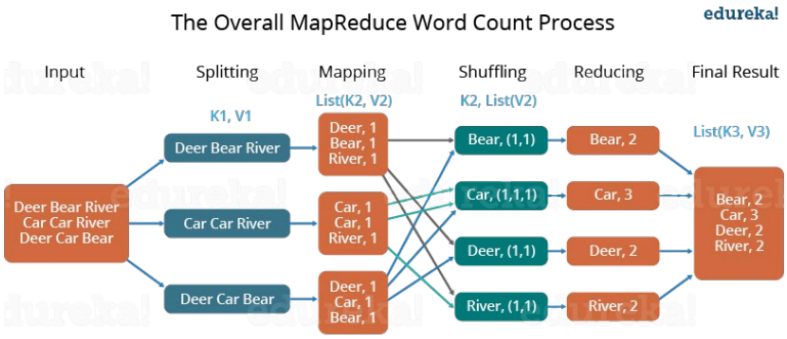
- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can't start until map phase is completely finished.

Transferring data with file system



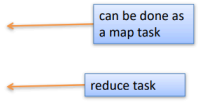
- Iterative MapReduce is required when you need to "sum up" results more than 1 times
- The MapReduce model is stateless, causing performance problems for iterative MapReduce programs

The Overall MapReduce Word Count Process



Expressing an Iteration in MapReduce

– for each point in the point set
  - find the closest mean I, assign the point to cluster I, forming new K clusters
– calculate new means for each cluster




```
map(point p):
    for each mean in K means
        find the closet mean M to this point
    emit_intermediate (index of M, p)

reduce(index of cluster i, list of points lp):
    calculate the new mean M' for points in lp
    emit(M')
```

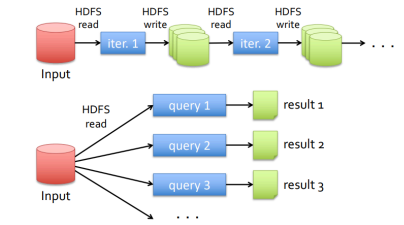
Why MapReduce is successful?

- Programmers only need to write serial code
  – Shared/private code in OpenMP
  – message orders in MPI
- Automatically parallel and distributed computing
- Automatic load-balancing and fault-tolerance

Limitations of MapReduce

- Only Map and Reduce
- Iterative MapReduce
  – Intermediate result on disk
  – Many I/O operations , poor performance
- Unable to support interactive query efficiently
  – No mechanism to share data in memory between MapReduce jobs

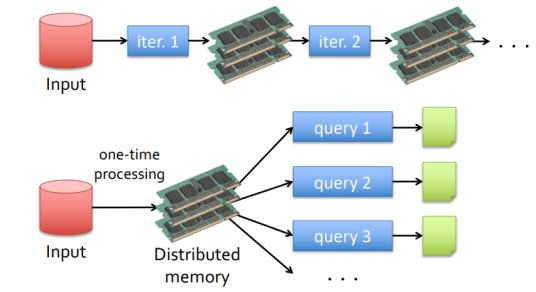
MapReduce Use File to Transfer data



The copy, serialization and disk I/O makes it slow

In Memory Computing

Use memory to store data



10-100 times faster than the disk approach

Issues

- Is memory large enough to contain all the data?
- How about the cost comparing to disks?
- Fault-tolerance with memory?
- How to represent the data in memory efficiently?

Two traditional abstractions

- Distributed Shared Memory
  - Extend the virtual memory

  read/write #address
- Distributed KV stores

  Value = Get(key)
  Put(key, value)

How to tolerate fault in distributed k-v stores?

A Log Sample

TID	T1	T1	T1	T2	T2	T2	T3
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50	COMMIT	A=80	B=70	COMMIT	A=110

```
begin  // T1
A = 100
B = 50
commit // A=100; B=50

begin  // T2
A = A - 20
B = B + 20
commit // A=80; B=70

begin  // T3
A = A + 30 // A=110
--CRASH--
```

Fault tolerance mechanism of DSM or Key-value stores

- Replica
  - High cost
  - Big time overhead
- Log
  - Big time overhead

Remote memory access
- 10-100 times slower than local memory accesses
- 1-10us vs 100ns

A more lightweight fault tolerance mechanism for in-memory computing is desired

Five Operations Involved

- **1. begin**: allocate a new transaction ID
- **2. write variable**: append an entry to the log
- **3. read variable**: scan the log looking for last committed value
  - As an aside: how to see your own updates?
  - Read uncommitted values from your own TID

Performance Problem of Log-only Approach

- **Write** performance is probably good
  - Sequential writes, instead of random
  - Need to write to disk *twice* instead of *once*
- **Read** performance is terrible!
  - Scan the log for every read!
- **Recovery** is instantaneous
  - Nothing to do

Five Operations Involved

- **4. Commit**: write a commit record
  - Expectedly, writing a commit record is the "commit point" for action, because of the way read works (looks for commit record)
  - However, writing log records better be all-or-nothing
    - One approach, from last time: make each record fit within one sector
- **5. Abort**: Do nothing or write a record?
  - Could write an abort record, but not strictly needed
- *Recover from a crash*: do nothing

Spark Programming

RDD (Resilient Distributed Datasets)

Solution

RDD（Resilient Distributed Datasets）
- Based on data sets, instead of single data
- Generated with deterministic coarse grained operations（map，filter，join etc.）
- Immutable
- To modify data, create new datasets by transformation

Efficient Fault Tolerance

- Once the data is generated deterministically, and immutable
  - Recover the data from "recompute"
  - Only need to log the sequences to generate rdd. small cost if there's no fault happens

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```

How to create RDDs

- From Collection
  val a = sc.parallelize(1 to 9, 3)
- from File
  val b = sc.textFile("/path/to/file")
- from other RDDs
  val c = b.map(line => line.split(","))

Scala - Functions

- def helloWord()={ println("Hello World")}
- def addOne(x:Int) = {
  x+1
  }
- def addOne(x:Int) = x+1
- val addOne = (x:Int) => x + 1

Spark Operators

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

RDD operations

- Transformation： generate new RDDs from existing RDDs
  - map, filter, groupBy, sort,distinct, sample…
- Action: return a value from RDD
  - reduce， count, collect, first, foreach…， ,

Example： WordCount

```
val spark = new SparkContext(master, appName, [sparkHome], [jars])
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Map

- Map each element in source RDD to ONE element in destination RDD

```
scala> val a = sc.parallelize(1 to 9, 3)
scala> val b = a.map(x => x*2)
scala> a.collect
res10: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
scala> b.collect
res11: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14, 16, 18)
```

reduceByKey

- Reduce for key-value list
- Values with the same keys are reduced, and form a new key-value

```
scala> val a = sc.parallelize(List((1,2),(3,4),(3,6)))
scala> a.reduceByKey((x,y) => x + y).collect
res7: Array[(Int, Int)] = Array((1,2), (3,10))
```

Use of _

- val a = (1 to 9)
- Val b = a.map(x => x*2)
  is the same as val b = a.map(_ * 2)
- val a = sc.parallelize(List((1,2),(3,4),(3,6))) a.reduceByKey((x,y) => x + y).collect is the same as a.reduceByKey(_+_).collect

flatMap

- One element in source rdd will be transformed to multiple elements(0..n) in destination rdd

```
scala> val a = sc.parallelize(1 to 4, 2)
scala> val b = a.flatMap(x => 1 to x)
scala> b.collect
res0: Array[Int] = Array(1, 1, 2, 1, 2, 3, 1, 2, 3, 4)
```

Join

(k,v1) join (k, v2) ➔ (k, (v1, v2))

```
scala> val kv1=sc.parallelize(List(("A",1),("B",2),("C",3)))
scala> val kv3=sc.parallelize(List(("A",4), ("A",5),("B",3),("D",30)))
scala> kv1.join(kv3).collect
  Array[(String, (Int, Int))] = Array((A,(1,4)), (A,(1,5)),(B,(2,3)))
scala> kv1.union(kv3).collect
  Array((A,1), (B,2), (C,3), (A,4), (A,5), (B,3), (D,30))
```

reduce

- Defines a function to combine two elements in a collection to one elements, and keep on applying it to elements in the collection

```
scala> val c = sc.parallelize(1 to 10)
scala> c.reduce((x, y) => x + y)
res4: Int = 55
```

Output

- saveAsTextFile/saveAsSequenceFile/saveAsObjectFile

```
scala> val a = sc.parallelize(List((1,2),(3,4),(3,6)))
scala> a.reduceByKey((x,y) => x + y).saveAsTextFile("test")
```

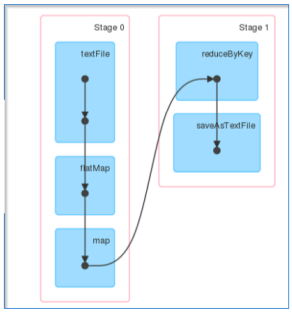
Join

(k,v1) join (k, v2) ➔ (k, (v1, v2))

```
scala> val kv1=sc.parallelize(List(("A",1),("B",2),("C",3)))
scala> val kv3=sc.parallelize(List(("A",4), ("A",5),("B",3),("D",30)))
scala> kv1.join(kv3).collect
  Array[(String, (Int, Int))] = Array((A,(1,4)), (A,(1,5)),(B,(2,3)))
scala> kv1.union(kv3).collect
  Array((A,1), (B,2), (C,3), (A,4), (A,5), (B,3), (D,30))
```

Programs represented by DAGs

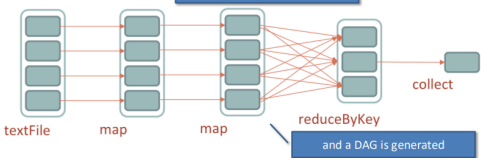
```
sparkContext.textFiles("hdfs://...")
        .map(line => line.split("\s"))
        .map(word => (word, 1))
        .reduceByKey((a, b) => a + b)
        .collect()
```



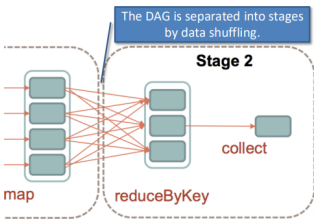
WordCount DAG(Partitioned RDDs)

```
sparkContext.textFiles("hdfs://...")          RDD[String]
        .map(line => line.split("\s"))        RDD[List[String]]
        .map(word => (word, 1))               RDD[(String, Int)]
        .reduceByKey((a, b) => a + b)         RDD[(String, Int)]
        .collect()                            Array[(String, Int)]
```

Action "collect" starts a new job

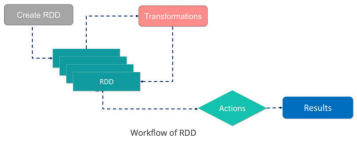


and a DAG is generated



How Spark evaluates RDD?

- Three level execution
  - Each action starts a new **job**
  - Each job is defined by a DAG of **stages**
  - Each stage contains multiple **tasks**
- Driver dynamically schedules pending tasks to executors



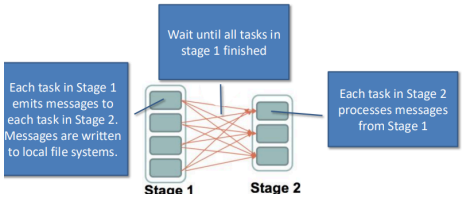
DAGs and Fault Tolerance

Rationale behind partition and lazy eval

- Lazy evaluation enables efficient fault tolerance
  - RDD's lineage-based fault tolerance, that log the sequences to generate RDD.
  - Lazy evaluation enables logging the sequences.
- Partition balances the failure recovery overhead
  - When an RDD partition is lost, re-compute the RDD partition.
  - It recomputes only the lost data.
  - Its recovery control overhead is small (elements are batched into partitions).

Data Shuffling

- Data shuffling is one of the fundamental and frequent operations in Spark.
- It generalizes MapReduce. A 2-stage job is exactly a MapReduce job.
- Data shuffling is expensive.

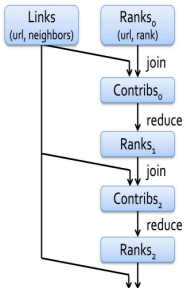


Data Partition Optimizations

Bye Rank

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

Eliminating Redundant Data Shuffling



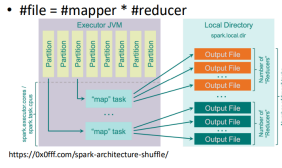
- Links and Ranks joins multiple times
- Partition the links and ranks with sample url together would avoid unnecessary communications
- partitionBy() are used to define how to partition data

How Spark performs Data Shuffling

- Currently Spark has implemented 4 methods
  - Hash Shuffle
    - Vanilla
    - ConsolidateFile
  - Sort Shuffle
  - Unsafe Shuffle

HashShuffle*

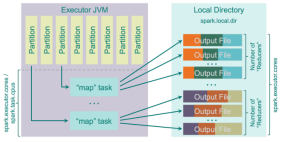
- #file = #mapper * #reducer



https://0x0fff.com/spark-architecture-shuffle/

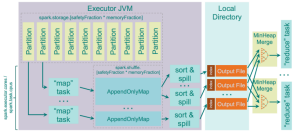
Hash Shuffle - ConsolidateFile

- #file = # of cores * #reducer



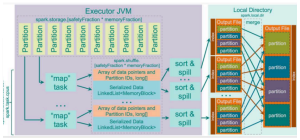
SortShuffle – sort and merge

- #file = #mapper, or more than that if spilled



Unsafe Shuffle

- sort elements in the form of raw byte arrays, thus more CPU-efficient



How Spark keeps and reuses RDD?

- RDDs can be kept in memory and reused in the future.
  - Avoid recalculation
  - Through ".cache()" or ".persist(...)" API call. "cache" is a special case for "persist"

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("bar")).count
```

How Spark keeps and reuses RDD? (cont.)

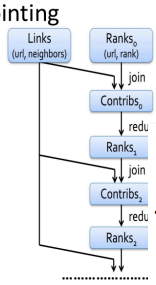
- Cached RDDs are distributedly stored in the block managers of executors.
- Tasks depending on cached RDDs will be preferably scheduled to containing executors to avoid extra communication.
- Missing partitions in a cached RDDs are OK. They will be re-evaluated according to the DAG.

Checkpointing

- DAG would get too long, causing too much recovery overhead.
  - Cached RDDs are not reliable.

Call ".checkpoint()" to store RDDs in a reliable storage media, such as HDFS.

After checkpointing, the RDD's DAG will be replaced by "reading from HDFS".



Data Shuffling Methods Summary

- Hash Shuffle has the least CPU overhead, but its #partitions is limited.
- Unsafe Shuffle almost has no #partitions limit but causes extra CPU overheads to sort.
- Sort Shuffle sorts elements in Java-object-form, enabling map-side combining, but causes high CPU overhead.

Limitations of Spark

- JVM based implementation
  - Memory size requirement(10X)
  - Garbage collection
    - Divide machines into many small containers
    - Each container contains 8-16GB memory
    - Out of memory error
    - Poor performance
- Solutions
  - Specialized systems such as graph computing systems
    - Following lectures
  - Efficient general purpose big data processing systems

Systems addressing Spark's performance issue

- New system from scratch in C++ (Husky, Thrill)
  - Toy system for proof-of-concept purpose: hard to deploy, missing functionalities, ...
  - Abandon Spark ecosystem: cloud, deployed clusters, ...
- Offloading UDF to native (SparkJNI)
  - Data marshalling overhead between the managed world and the native world
  - Complicated to program and maintain
- Higher abstraction level (Tungsten, Flare)
  - Restricted to certain application domains
  - Lack of general-purpose programming model

hukonu: A Fully-Featured High-Performance Big Data
nework that Integrates a Native Compute Engine into Spark[1]

roviding easily-accessible C++ API (nearly identical to park API)
- Familiar to Apache Spark users

nabling bare-metal performance
- Lower query latency
- Saves computing resources

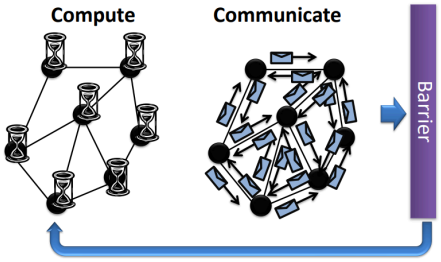
ntegrating into Apache Spark with little overhead
- Reuses Hadoop/Spark ecosystem
- Reuses Spark's production-grade functionalities

Graph Analysis

Spark isn't suited for extreme-scale graph processing

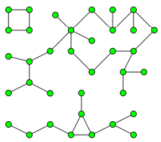
- Slow and memory inefficient

The BSP model does not support asynchronous computing



Weakly Connected Components

- Algorithm
  - Give each vertex an Look at all edges and find the vertex id who holds the highest value.
  - Send the greatest vertex id to all the edges.
  - Process incoming messages and compare it against the value you hold.
  - Repeat the process until there are no messages being sent.



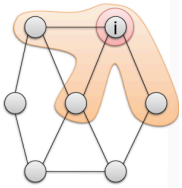
What happens if this algorithm is executed asynchronously?

Asynchronous processing

The GraphLab Abstraction

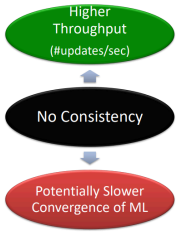
Vertex-Programs directly **read** the neighbors state

```
GraphLab_PageRank(i)
  // Compute sum over neighbors
  total = 0
  foreach( j in in_neighbors(i)):
    total = total + R[j] * w_ji

  // Update the PageRank
  R[i] = 0.15 + total

  // Trigger neighbors to run again
  if R[i] not converged then
    foreach( j in out_neighbors(i)):
      signal vertex-program on j
```

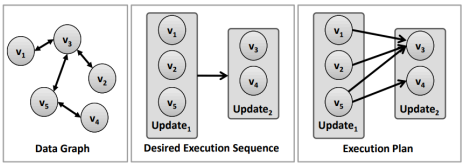


Ensuring Race-Free Code in Async processing:

Need for Consistency?

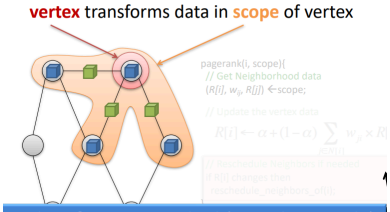


An execution plan example



Update Functions

User-defined program: applied to **vertex** transforms data in **scope** of vertex



Update function applied (asynchronously) in parallel until convergence

Graph partitioning on power law graphs and GAS model

Needed because Graphs can become unhandable huge.

Random Partitioning

- Both GraphLab 1 and Pregel proposed Random (hashed) partitioning for Natural Graphs

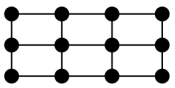
For *p* Machines:

$$\mathbb{E}\left[\frac{|Edges\ Cut|}{|E|}\right] = 1 - \frac{1}{p}$$

10 Machines → 90% of edges cut
100 Machines → 99% of edges cut!

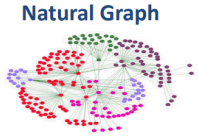
Assumptions of Graph-Parallel Abstractions

Idealized Structure



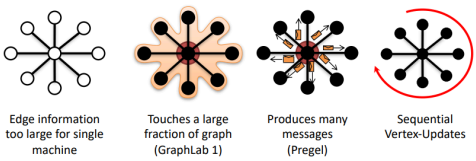
- ***Small*** neighborhoods
  - Low degree vertices
- Vertices have similar degree
- Easy to partition

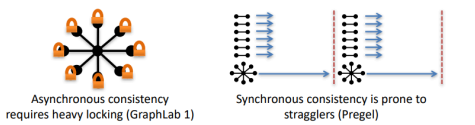
Natural Graph



- ***Large*** Neighborhoods
  - High degree vertices
- ***Power-Law*** degree distribution
- ***Difficult to partition***

Problem:
High Degree Vertices Limit Parallelism





In Summary

GraphLab 1 and Pregel are not well suited for natural graphs

- Poor performance on high-degree vertices
- Low Quality Partitioning

GraphLab2

- Distribute a single vertex-update
  - Move computation to data
  - Parallelize high-degree vertices
- Vertex Partitioning
  - Simple online approach, effectively partitions large power-law graphs

Minimizing Communication in **GraphLab 2: Vertex Cuts**

A **vertex-cut** minimizes # machines per vertex

Percolation theory suggests Power Law graphs can be split by removing only a small set of vertices [Albert et al. 2000]

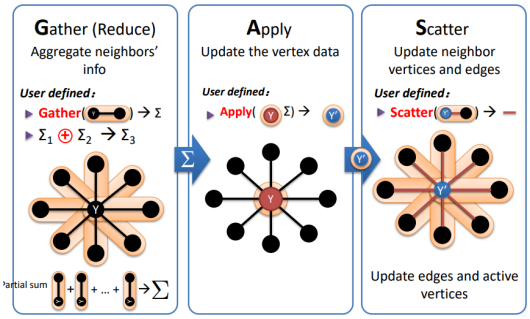
➔

Small vertex cuts possible!

A **Common Pattern** for Vertex-Programs

```
GraphLab_PageRank(i)
  // Compute sum over neighbors
  total = 0
  foreach( j in in_neighbors(i)):
    total = total + R[j] * w_ji        Gather Information About Neighborhood

  // Update the PageRank
  R[i] = 0.1 + total                   Update Vertex

  // Trigger neighbors to run again
  if R[i] not converged then
    foreach( j in out_neighbors(i))    Signal Neighbors & Modify Edge Data
      signal vertex-program on j
```

GAS



PageRank in PowerGraph

$$R[i] = 0.15 + \sum_{j \in \mathrm{Nbrs}(i)} w_{ji} R[j]$$

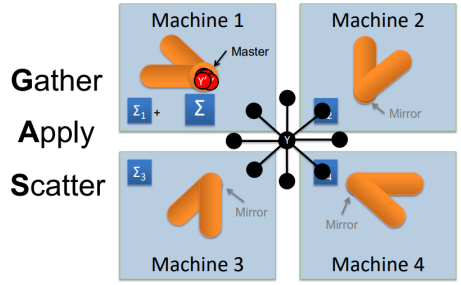
PowerGraph_PageRank(i)

Gather(j → i) : return $w_{ji} * R[j]$
sum(a, b) : return a + b;

Apply(i, Σ) : $R[i] = 0.15 + \Sigma$

Scatter(i → j) :
if *R[i]* changed then trigger *j* to be **recomputed**

Distributed GAS



Constructing Vertex-Cuts

- **Goal:** *Parallel graph partitioning on ingress*
- GraphLab 2 provides three **simple** approaches:
  - ***Random*** *Edge Placement*
    - Edges are placed randomly by each machine
      - Good theoretical guarantees
  - ***Greedy*** *Edge Placement with* ***Coordination***
    - Edges are placed using a shared objective
      - Better theoretical guarantees
  - ***Oblivious-Greedy*** *Edge Placement*
    - Edges are placed using a local objective

PowerLyra

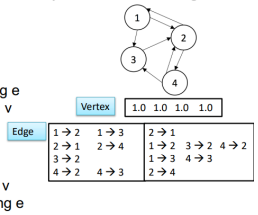
- PowerGraph partitions graph for high degree vertices, but is not friendly to the low degree vertices(locality)
- Optimize the low degree vertices' locality
  - Avoids unnecessary communications

Out of core graph processing

The key of out-of-core graph systems is the sequential access to graph data

Generalized Graph Processing

```
for each vertex v
  if v is active
    for each edge e to v
      gather update along e
  apply updated value to v

for each vertex v
  if v has update
    for each edge e from v
      scatter update along e
```

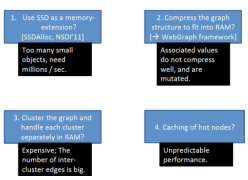


Can we use just one copy of the edge list?
Do we need all vertices in memory?

Out-of-core Graph Processing

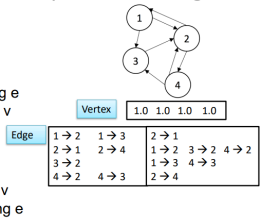
- Processing large graphs with limited memory
  - Low cost graph analysis engine

Possible Solutions



Generalized Graph Processing

```
for each vertex v
  if v is active
    for each edge e to v
      gather update along e
  apply updated value to v

for each vertex v
  if v has update
    for each edge e from v
      scatter update along e
```



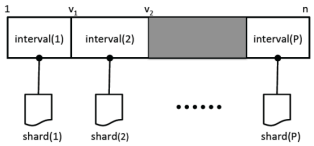
Gather&Apply can be separated from Scatter

GRAPHCHI – Parallel Sliding Window

PSW: Shards and Intervals

1. Load
2. Compute
3. Write

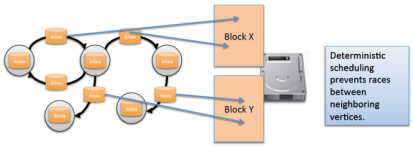
- Vertices are numbered from 1 to n
  - **P** intervals, each associated with a **shard** on disk.
  - **sub-graph =** interval of vertices



PSW: Execute updates

1. Load
2. Compute
3. Write

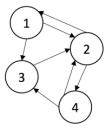
- Update-function is executed on interval's vertices
- Edges have pointers to the loaded data blocks
  - Changes take effect immediately → asynchronous.



Graphchi processing model

```
for each vertex partition (Intervals)
    load all vertex and in/out edges

    for each vertex v
      if v is active
        for each edge e to v
          gather update along e
      apply updated value to v

    for each vertex v
      if v has update
        for each edge e from v
          scatter update along e by attaching the info

    Write all in/out edges
```

Example

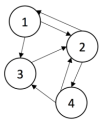


Interval	1	2	3	4

Shard		
1 → 2 2 → 1 3 → 2 4 → 2		1 → 3 2 → 4 4 → 3

How is gather and scatter implemented?

Example with (simplified) PageRank



Interval	1.0	1.0	1.0	1.0

Shard		
1 → 2 0.5 2 → 1 0.5 3 → 2 1.0 4 → 2 0.5		1 → 3 0.5 2 → 4 0.5 4 → 3 0.5

GraphChi → XStream → GridGraph

Dual sliding window to allow sequential access to edges

Remove the need to sort in preprocessing

2-D partition to allow selective scheduling on edges; In place apply to avoid shuffle

PSW: Commit to Disk

1. Load
2. Compute
3. Write

- In write phase, the blocks are written *back* to disk
  - Next load-phase sees the preceding writes → asynchronous.

In total:
P² reads and writes / full pass on the graph.
→ Performs well on *both* SSD and hard drive.

GraphChi is good

- But have two major drawbacks
  - P^2 I/O are required
    - Poor performance when P is large
  - High preprocessing time because sorting is required for each shard

XSTREAM – Edge Centric

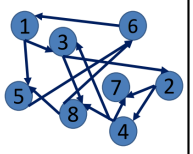
X-Stream : Edge-Centric model

for each vertex v if v has update for each edge e from v scatter update along e	Scatter

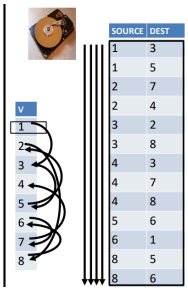
for each edge e If e.src has update scatter update along e	Scatter

Vertex-Centric ⇨ **Edge-Centric**

Edge Centric Processing



BFS



Tradeoff

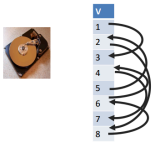
Vertex-centric Scatter-Gather: *Edge Data/ Random Access Bandwidth*

- Edge-centric Scatter-Gather
  - #of scatters*Edge data/sequential access bandwidth
- Sequential Access Bandwidth >> Random Access Bandwidth
- Few scatter-gather iterations for real world graphs
  - Give an example that is not suitable for this system?

Streaming Partitions

Streaming Partitions

- Problem: still have random access to vertex set



- Solution: partition the graph into streaming partitions

Streaming Partitions

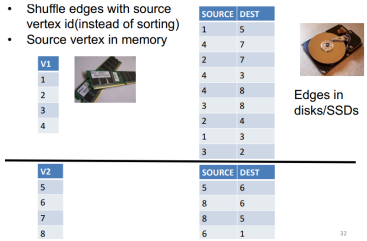
- A streaming partition is
  - A subset of the vertices that fits in RAM
  - All edges whose source vertex is in that subset
  - No requirement on quality of the partition

The issues of X-Stream

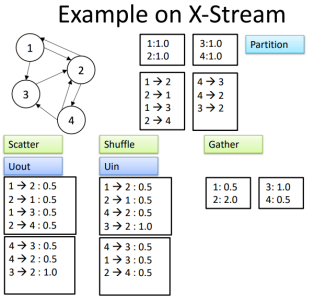
- X-Stream
  - Separate gather/scatter phase, with shuffle between them, introduce many extra I/O operations
  - Poor performance for BFS and WCC-like algorithms
    - Only a small portion of edges are touched for some iterations, but they would scan all edges

Partition Edges and vertexs

- Shuffle edges with source vertex id(instead of sorting)
- Source vertex in memory



Example on X-Stream



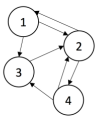
GRIDGRAPH – 2D Partition

GridGraph

- A 2-level partitioning graph data structure
  - Improve locality
  - Enable more effective streaming on both edges and vertices
- A Streaming-Apply model
  - Combines the gather-apply-scatter operations
- A programming abstraction
  - To enable the streaming-apply model

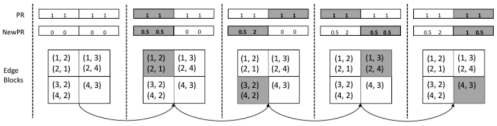
The Grid Representation

- 2-level hierarchical partitioning
  - 1st dimension
    - partition the vertices into chunks(1 2)(3,4)
    - partition the edges into shards by source vertex((1,2),(2,1),(1,3),(2,4))
  - 2nd dimension
    - partition the shards into blocks by destination vertex



(1, 2) (2, 1)	(1, 3) (2, 4)
(3, 2) (4, 2)	(4, 3)

The Streaming-Apply Model



Dual Sliding Windows:
- 1 read window + 1 write window
- Contents of active windows can be fit into RAM

Each iteration, just read edge list once, has opportunity to skip some locks

Algorithm 1 Edge Block Streaming

```
for j ← 1, P do
    for i ← 1, P do
        if ChunkIsActive(i) then
            StreamEdgeBlock(i, j);
        end if
    end for
end for
```

Streaming-Apply Processing Model

- Vertex accesses are aggregated
  - Good locality
    - Only 2 partitions of vertices are accessed within each edge block
- On-the-fly updates onto vertices
  - Reduces I/O
  - Enables asynchronous implementation of algorithms (like WCC, etc.) which converges faster

GridGraph processing model

for each vertex partition
    load all vertex and in/out edges

    for each edge e
        gather update along e
        read e.src
        apply updated value to e.dst

Write the unnecessary vertex partition back to disk

Programming Abstraction

Algorithm 2 Edge Streaming Interface

```
function StreamEdges(F_e, F)
    Sum = 0
    for all block do                    ▷ with active vertices
        for all edge ∈ block do
            if F(edge.source) then
                Sum += F_e(edge)
            end if
        end for
    end for
    return Sum
end function
```

← StreamEdges(Fe, F):
Fe: process the edge, which is allowed to side-effect related vertex data
F: check whether the edge should be processed (by looking at the source vertex)

→ StreamVertices(Fv, F):
Fv: process the vertex
F: check if the vertex should be processed

Algorithm 3 Vertex Streaming Interface

```
function StreamVertices(F_v, F)
    Sum = 0
    for all vertex do
        if F(vertex) then
            Sum += F_v (vertex)
        end if
    end for
    return Sum
end function
```

BFS

Algorithm 4 BFS

```
function IsActive(v)
    return ActiveIn[v]
end function
function Visit(e)
    if CAS(&Parent[e.dest], -1, e.source) then
        ActiveOut[e.dest] = 1
        return 1
    end if
    return 0
end function
ActiveVertices = 1
Parent = {-1, ..., -1}
ActiveIn = {0, ..., 0}
Parent[start] = start
ActiveIn[start] = 1
while ActiveVertices > 0 do
    ActiveOut = {0, ..., 0}
    ActiveVertices = StreamEdges(Visit, IsActive)
    Swap(ActiveIn, ActiveOut)
end while
```

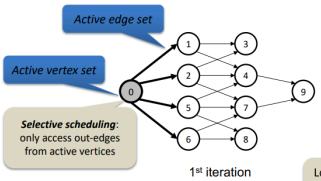
PageRank Implementation

Algorithm 3 PageRank

```
function Contribute(e)
    Accum(&NewPR[e.dest], PR[e.source] / Deg[e.source])
end function
function Compute(v)
    NewPR[v] = 1 − d + d × NewPR[v]
    return |NewPR[v] − PR[v]|
end function
d = 0.85
PR = {1, ..., 1}
Converged = 0
while ¬Converged do
    NewPR = {0, ..., 0}
    StreamEdges(Contribute)
    Diff = StreamVertices(Compute)
    Swap(PR, NewPR)
    Converged = Diff / V ≤ Threshold
end while
```

Top Down vs. Bottom Up

BFS Example （top-down model)



Locks/atomic operations required for correctness of concurrent updates

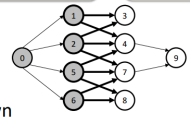
```
function top-down-step(vertices, frontier, next, parents)
    for v ∈ frontier do
        for n ∈ neighbors[v] do
            if parents[n] = -1 then
                parents[n] ← v
                next ← next ∪ {n}
            end if
        end for
    end for
```

Heuristics for mode switch

Dual mode updates proposed in shared-memory systems (Ligra[PPoPP '13])

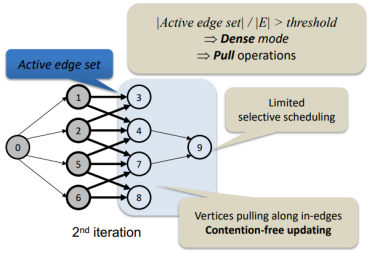
|Active edge set| / |E| < threshold
⇒ **Sparse** mode
⇒ **Push** operations

|Active edge set| / |E| > threshold
⇒ **Dense** mode
⇒ **Pull** operations



- Top-down
  - Less active edges – sparse – push
- Bottom-up
  - More active edges – dense – pull

BFS(Bottom-up mode)



Bottom-up BFS

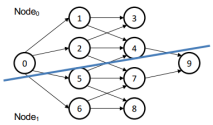
```
function bottom-up-step(vertices, frontier, next, parents)
    for v ∈ vertices do
        if parents[v] = -1 then
            for n ∈ neighbors[v] do
                if n ∈ frontier then
                    parents[v] ← n
                    next ← next ∪ {v}
                    break
                end if
            end for
        end if
    end for
```

GEMINI – Distributed push/pull

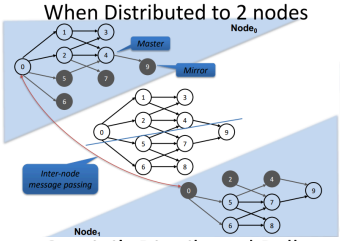
We Propose: ***Gemini***

- Build **scalability** on top of **efficiency**
  - Avoid unnecessary "distributed" side-effects
  - Optimize computation on partitioned graphs
- Shift of design focus
  - Designed for distributed, but **computation-centric**
    - Modern clusters have fast interconnects
    - Computation-communication overlap in place
  - Efficiency optimizations
    - Adaptive push-/pull-style computation
    - Hierarchical chunk-based partitioning
  - Scalability optimizations
    - Locality-aware chunking
    - Chunk-based work-stealing

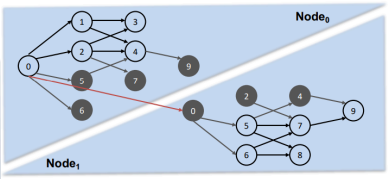
Distributed Dual-mode Computation



When Distributed to 2 nodes

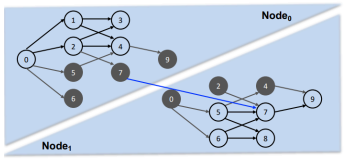


Gemini's Distributed Push



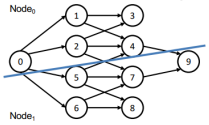
Masters message mirrors, who update their local neighbors

Gemini's Distributed Pull

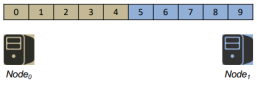


Mirrors pull updates from neighbors, then message masters

Gemini's Choice for Graph Partition



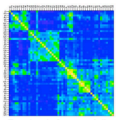
- Chunking
  - Divide vertex set ***V*** into ***p*** **contiguous** chunks



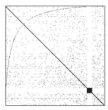
  - Dual-mode edge data distributed accordingly

Why Chunk-based Partitioning?

- It preserves locality!
  - Fact: **locality** exists in many real-world graphs
    - E.g., WebGraph[WWW '04], BLP[WSDM '13]
    - Vertices "semantically" ordered
    - Small distance between source and destination vertices



Facebook Country Adjacency Matrix[1]



UK Web (2005) Adjacency Matrix

  - Preprocessing affordable when vertices unordered
    - E.g., BFS[Algorithms 09], LLP[WWW '11]

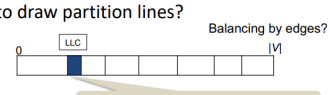
More Benefits of Chunking

- No need to convert vertex IDs (global ⇔ local)
  - Simple bookkeeping of partition information
    - O(*p*) chunk boundaries
  - Simple management of vertex data arrays
    - Just allocate entire array in shared memory

- Can be applied recursively at different levels

Locality-aware Chunking

- Where to draw partition lines?

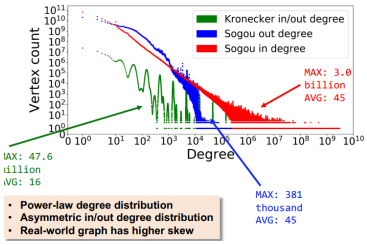


Chunk size affects random access efficiency!

- Gemini considers both vertex and edge
  - Edge: the amount of work to be processed
  - Vertex: the processing speed of work (locality)
  - Hybrid metric: $\alpha \cdot |V_i| + |E_i|$
    - α set to 8(p-1) currently as default value

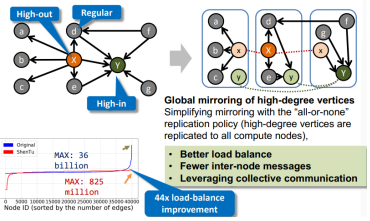
Shentu - Designed for Extreme Scale Graph

Problem 1: Severe Load Imbalance



- **Power-law degree distribution**
- **Asymmetric in/out degree distribution**
- **Real-world graph has higher skew**

Solution 1: Degree Aware Messaging

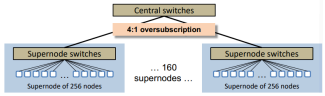


Global mirroring of high-degree vertices
Simplifying mirroring with the "all-or-none" replication policy (high-degree vertices are replicated to all compute nodes),

- **Better load balance**
- **Fewer inter-node messages**
- **Leveraging collective communication**

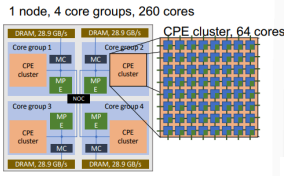
Problem 2a: Too Many Small Messages

- Traditional distributed graph computing model (1D)
  - $O(N \times N)$ messages
  - Not suitable to extreme-scale processing
    - Too expensive on 40,000+ nodes
    - # nodes >> average degree: little opportunity for combining messages
- Hierarchical interconnect



Problem 3: Heterogeneous Architecture

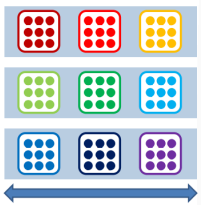
- MPE (Management Processing Element)
  - General-purpose processor
  - Low memory bandwidth
- CPE (Compute Processing Element)
  - High memory bandwidth
  - Limited functionality
- Shared memory between MPE and CPE



Requires careful task mapping and coordination

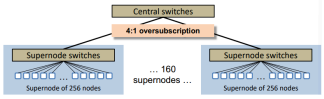
Solution 2a: Supernode Routing

- Stage 1
  - Messages grouped by destination supernodes
  - Combined messages sent to relay node in destination supernode
- Stage 2
  - Messages grouped by destination nodes within same supernode
  - Combined messages sent to final destination node

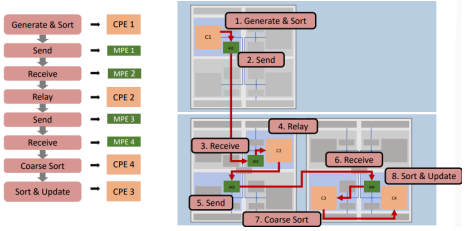


Problem 2a: Too Many Small Messages

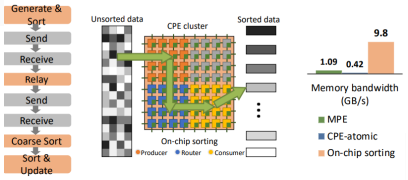
- Traditional distributed graph computing model (1D)
  - $O(N \times N)$ messages
  - Not suitable to extreme-scale processing
    - Too expensive on 40,000+ nodes
    - # nodes >> average degree: little opportunity for combining messages
- Hierarchical interconnect



Solution 3a: Heterogeneous Task Mapping

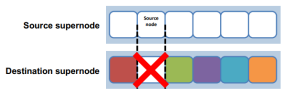


Solution 3b: CPE Cluster On-chip Sorting



Solution 2b: Topology-aware Super-node Relaying

Failed nodes forbid direct source-relay mapping



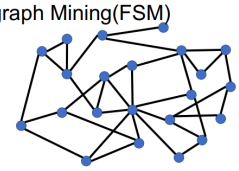
Stochastic assignment: load balancing in destination supernode

Graph Mining

Graph Mining

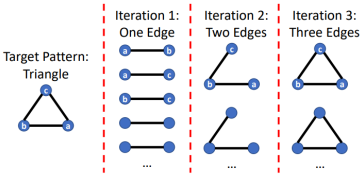
- Find given patterns in graph
  - Triangle, Clique
    - Count numbers or output all subgraphs
- Frequent Subgraph Mining(FSM)


Pattern

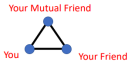

Dataset

Problems of TLE

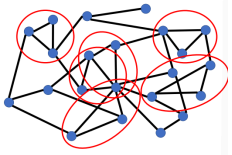
- BFS style embedding extension
  - Huge intermediate states (exponentially increased with number of edges)



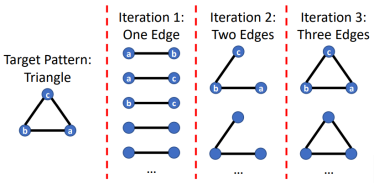
Triangle Counting



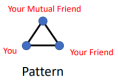
Pattern


Dataset

Think Like a Embedding(TLE)



Triangle Counting is well studied



Pattern

Algorithm 1: Triangle counting.

input : G : the Graph.
output: n : the number of triangles in G.

```
1 begin
2     n ← 0;
3     for v0 in V do
          // N(v) returns a set that contains
             all v's neighbors
4         for v1 in N(v0) do
5             s ← N(v0) ∩ N(v1);
6             n ← n + |s|;
7     n ← n/6;
```

Algorithm 1: Exploration step in Arabesque.

input : Set I of initial embeddings for this step
output: Set F of extended embeddings for the next step (initially empty)
output: Set O of new outputs (initially empty)

foreach $e \in I$ *such that* $\alpha(e)$ **do**
  add $\beta(e)$ to O;
  $C \leftarrow$ set of all extensions of e obtained by adding one incident edge / neighboring vertex;
  foreach $e' \in C$ **do**
    if $\phi(e')$ *and there exists no* $e'' \in F$ *automorphic to* e' **then**
      add $\pi(e')$ to O;
      add e' to F;
run aggregation functions;

Stream Processing

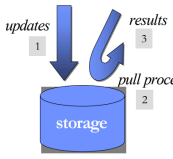
What we did so far: Batch processing, processing all data

- Batch processing, processing all data
  - Map-Reduce
  - Spark
  - GraphLab / Gemini

Stream Processing

- *F(X +ΔX)= F(X) op H(ΔX)*
- When we calculate *F(X +ΔX) , we don't need to use X +ΔX*， instead we only need F(X) and *ΔX*
- We call this computing paradigm Stream Processing

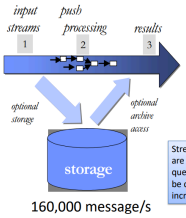
Use traditional database



Databases are good for adhoc queries

900 message/s

Stream Processing



Stream processing are good at fixed queries which can be calculated incrementally

160,000 message/s

Real-time / Scalability

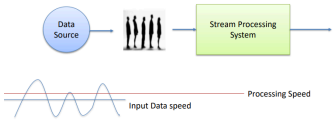
- Batch processing
  - Fixed problem size
  - Latency: Tens of hours processing time
- Stream Processing
  - Arrival rate varies heavily
  - Latency: Seconds or even less
  - Processing all coming data or
  - Pre-defined degrading approach

Fault Tolerance

- Batch
  - ETL can remove faults in data
  - Checkpoint and/or recalculation
- Streaming
  - Has to deal with data fault in real time
  - Fault tolerance mechanism must be low-overhead, and maintain the real time demand

Naive Implementation

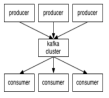
Mismatched Speed of Producer/Consumer : Add a Queue



Queue is used to address the TEMPORAL speed mismatch

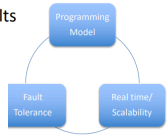
Queue

- Matching Producers and Consumers
  - Speed matching
  - Interfaces
    - Publish – Subscribe model
  - Persistency
    - Could replay data
  - Performance optimization
    - Batching data(Similar to relay in Shentu)



Problems

- Poor scalability
- Hard to tolerate faults
- Complicated code

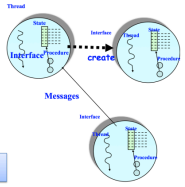


Actor Model

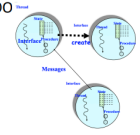
Actor Model

- An actor
  - An asynchronous concurrent programming paradigm
- An actor may
  - Process messages
  - Change local state
  - Create new actors
  - Send messages

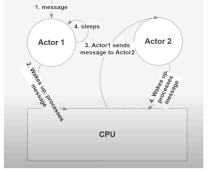
Similar to MPI, but more dynamic, asynchronous and finer grained



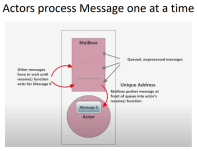
- Key ideas
  - Everything is an Actor
  - Analogous to "everything is an object" in OO
  - Mutable state is not shared
- Other important features
  - Transparent addressing
  - Asynchronous message passing
  - Non-deterministic ordering of messages
  - No assumption on channel
  - Unbounded delay
  - Best effort message delivery
  - Only one message is processed at a time conceptually



Actor run asynchronously*



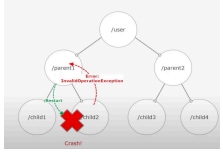
Actors process Message one at a time



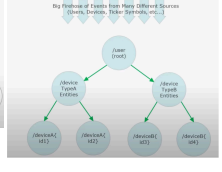
Comparison Actor models with Queue Systems

	Actor Model (akka)	Queue (Kafka)	MPI
Messaging modes	Asynchronous	Asynchronous (Pub/Sub)	Synchronous / Asynchronous
Latency	~1 us	~100ms	~1 us
Persistency	N	Y	N
Purpose	Building concurrent systems, such as distributed databases, distributed streaming systems	Connecting different systems	Building parallel computing applications, such as scientific engineering computing applications(WRF, VASP etc.) and LLM training systems

Actors in hierarchy



Actors reduce big problems into smaller ones



Stream Processing – STORM

Core Concepts and Architecture

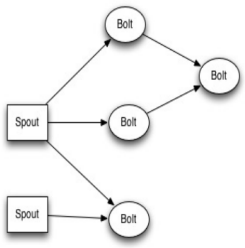
1. **Stream**:
   - The core abstraction in Storm is the "stream," which is an unbounded sequence of tuples (data records).
   - Streams are the primary input/output data format for all operations in a Storm topology.
2. **Tuple**:
   - A tuple is a named list of values, essentially the smallest unit of data processed in a Storm application.
   - Tuples can contain any number of fields, such as integers, strings, or objects.
3. **Topology**:
   - A topology is a directed acyclic graph (DAG) that represents the flow of data between computations in Storm.
   - A topology runs indefinitely until terminated, processing streams of data in real-time.
4. **Spouts**:
   - Spouts are the sources of streams in a Storm topology.
   - They emit data tuples into the topology, typically pulling data from external sources like message queues (e.g., Kafka), databases, or APIs.
5. **Bolts**:
   - Bolts process tuples from spouts or other bolts. They are the building blocks for processing logic.
   - Bolts can perform various tasks such as filtering, aggregation, joining streams, or interacting with databases.
6. **Stream Grouping**:
   - Determines how tuples are routed between bolts.
   - Examples:
     - **Shuffle Grouping**: Distributes tuples randomly.
     - **Fields Grouping**: Routes tuples to bolts based on specific fields.
     - **All Grouping**: Sends each tuple to all bolts.
     - **Global Grouping**: Sends all tuples to a single bolt.

Key Features

1. **Fault Tolerance**:
   - Storm ensures reliability through an acknowledgment system. If a tuple fails to process, it can be replayed.
   - Worker nodes can restart automatically in case of failure.
2. **Scalability**:
   - Storm can handle a high throughput of messages by parallelizing computation.
   - Parallelism is achieved by specifying the number of executors and tasks for spouts and bolts.
3. **Low Latency**:
   - Designed for sub-second processing times, Storm is ideal for applications requiring immediate insights.
4. **Language Agnostic**:
   - Storm's components can be written in any programming language that supports JSON serialization.
5. **Integration**:
   - Storm integrates well with other technologies like Kafka (as a spout), HDFS (for long-term storage), and databases (for persistence).

Stream Grouping

- Shuffle Grouping：Randomly
- Fields Grouping：Choose according to tuple field's value
- All Grouping：all tasks
- Global Grouping：send to tasks with the minimum id

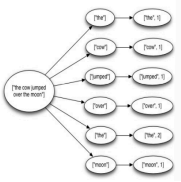


Fault tolerance in Stream Processing

- Per element processing(at least once) - Storm
- Mini-batch(Exactly once) – SparkStream
- Per element processing(Exactly once) - Flink

Complete processing of a message

- When spouts emit tuples, specify a unique message ID.
- When bolts are done processing the input tuple, ack or fail the input tuple.



Running Storm

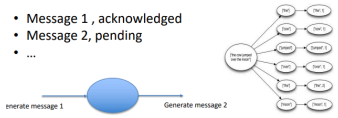
```
LocalCluster cluster = new LocalCluster();

Map conf = new HashMap();
conf.put(Config.TOPOLOGY_DEBUG, true);

cluster.submitTopology("demo", conf, builder.createTopology());
```

How to track if the message has been processed?

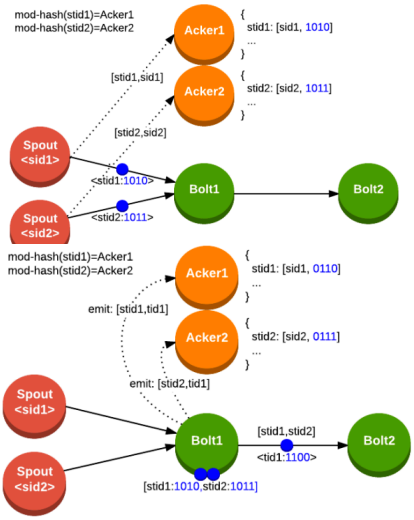
- We can follow the message flow to see if there are outstanding messages
- Use a data structure to record the generations/acknowledgement of messages
  - Do we need a new entry for each of generated message?

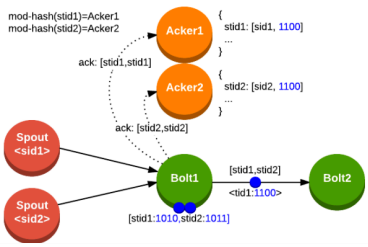
Maintain the pending message

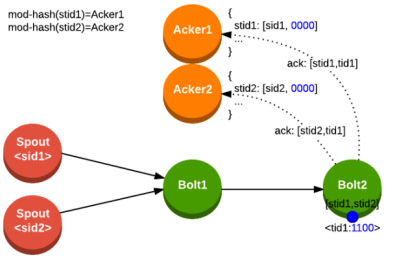
- Message 1 , acknowledged
- Message 2, pending
- …



- High memory overhead, also does not know who is the root message of them
- A better way is to have an entry for each source message







1 Define the spout

TestWordSpout

```
public void nextTuple()
{ Utils.sleep(100);
final String[] words = new String[] {"nathan", "mike", "jackson", "golda", "bertels"};
final Random rand = new Random();
final String word = words[rand.nextInt(words.length)];
_collector.emit(new Values(word));
}
```

emit() is to push the data to downstream bolts

3. Define the topology

ExclamationTopology

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("words", new TestWordSpout(), 10);
builder.setBolt("exclaim1", new ExclamationBolt(), 3) .shuffleGrouping("words");
builder.setBolt("exclaim2", new ExclamationBolt(), 2) .shuffleGrouping("exclaim1");
```

WordCount in Storm

```
public static class WordCount implements IBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();

    public void prepare(Map conf, TopologyContext context) {
    }

    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if(count==null) count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }

    public void cleanup() {
    }
```

2. Define the bolts

ExclamationBolt

```
public void execute(Tuple tuple) {
  _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
  _collector.ack(tuple);
}

public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("word"));
}
```

Notice the ack() here, why it is necessary?

SparkStream

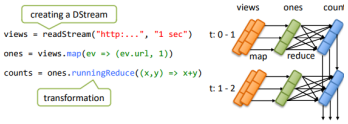
Review: How Spark deal with Fault Tolerance

- Computation are implemented as deterministic transformations on immutable data sets

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```

Example: Counting page views

Discretized Stream (DStream) is a sequence of immutable, partitioned datasets
- Can be created from live data streams or by applying bulk, parallel transformations on other DStreams

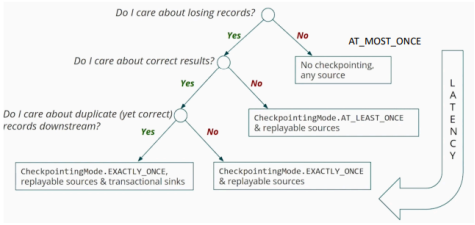


SparkStream WordCount code example*

```
// Create a ReceiverInputDStream on target ip:port and count the
// words in input stream of \n delimited test (e.g. generated by 'nc')
val lines = ssc.socketTextStream(args(0), args(1).toInt)
val words = lines.flatMap(_.split(" "))
val wordDstream = words.map(x => (x, 1))

// Update the cumulative count using mapWithState
// This will give a DStream made of state (which is the cumulative count of the words)
val mappingFunc = (word: String, one: Option[Int], state: State[Int]) => {
  val sum = one.getOrElse(0) + state.getOption.getOrElse(0)
  val output = (word, sum)
  state.update(sum)
```

Different fault tolerance levels

At most once, At least once, Exactly once



Fault tolerance in Stream Processing

- Per element processing(at least once) - Storm
- Mini-batch(Exactly once) – SparkStream
- Per element processing(Exactly once) - Flink

Discretized Stream Processing

Run a streaming computation as a series of small, deterministic batch jobs

Store intermediate state data in cluster memory

Try to make batch sizes as small as possible to get second-scale latencies

Comparison of different Streaming Models

Comparison of different systems

- Programming model
  - Storm and Flink requires save and processing more
  - SparkStream is an extension of Spark
  - But Flink provide SQL like interface which is very user friendly

Flow control

- Push or pull
  - Spark uses push
  - Storm and Flink uses pull, buffer overflow only at the first stage
    - Pull is also known as back pressure

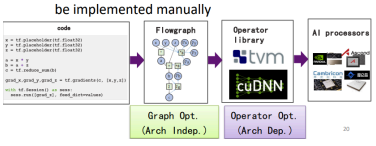
Fault tolerance

- Fault Tolerance
  - Storm guarantees a message is processed (at least) once
  - SparkStream uses mini-batch to provide exact-once
  - Flink (and storm trident) can do per-message exactly once with checkpoint, with replayable source(kafka) and transactional sink

Machine Learning

Machine Learning Systems

- Compile the program to flowgraph and optimize it
- Support libraries for new hardware
  - Thousands of operators which are not affordable to be implemented manually



Matrix Multiplication Loop Orders

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

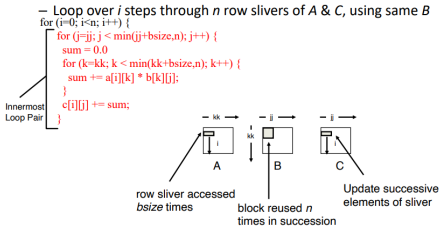
```
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

```
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C*
- Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*



Halide

Halide

- Halide is a domain specific language
- It decoupled the optimized implementation

Program = algorithm + scheduling

  - Enables programmers with different skills to collaborate
    - Algorithm experts and system experts
  - Compilers are used to generate the parallel and optimized code
- Originally designed for image processing,
  - Now used by AI compilers for AI operator generation

Tensor-based abstraction

- `C = matrix_multiply(A, B.T)`

```
m, n, h = tvm.var('m'), tvm.var('n'), tvm.var('h')
A = tvm.placeholder((m, h), name='A')
B = tvm.placeholder((n, h), name='B')

k = tvm.reduce_axis((0, h), name='k')
C = tvm.compute((m, n),
    lambda i, j: tvm.sum(A[i, k] * B[j, k], axis=k)
)
```

Inp

Output matrix shape

Computation rule:

$$C[i,j] = \sum_k A[i,k] \times B[j,k]$$

More tensor expressions

- Affine transformations

```
out = tvm.compute((n,m), lambda i, j: tvm.sum(data[i,k] * w[j,k], k))
out = tvm.compute((n,m), lambda i, j: out[i,j] + bias[i])
```

- Convolution

```
out = tvm.compute((c, h, w),
    lambda i, x, y: tvm.sum(data[kc,x+kx,y+ky] * w[i,kx,ky], [kx,ky
,kc]))
```

- ReLU

```
out = tvm.compute(shape, lambda *i: tvm.max(0, out(*i)))
```

Scheduling example

```
C = tvm.compute((n, ), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
```

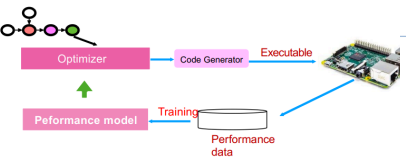
Scheduling example

```
C = tvm.compute((n, ), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
```

Scheduling example

```
C = tvm.compute((n, ), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
s[C].reorder(xi, xo)
s[C].bind(xo, tvm.thread_axis("blockIdx.x"))
s[C].bind(xi, tvm.thread_axis("threadIdx.x"))
```

Machine learning based optimizer



The machine learning model predict the performance within 1ms

```
for (int i = 0; i < n; i++) {
    C[i] = A[i] + B[i]
}
```

```
for (int xo = 0; xo < ceil(n / 32); ++xo) {
    for (int xi = 0; xi < 32; ++xi) {
        int i = xo * 32 + xi;
        if (i < n) {
            C[i] = A[i] + B[i];
        }
    }
}
```

```
int i = threadIdx.x * 32 + blockIdx.x;
if (i < n) {
    C[i] = A[i] + B[i];
}
```

Graph Optimization

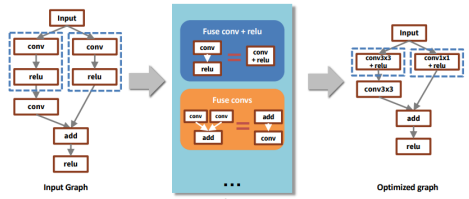
Fusion of operators


```
for( i in 1:n )
  tmp1[i,1] = s * B[i,1];
for( i in 1:n )
  tmp2[i,1] = A[i,1] + tmp1[i,1];
for( i in 1:n )
  R[i,1] = tmp2[i,1] * C[i,1];
```

```
for( i in 1:n )
  R[i,1] = (A[i,i] + s*B[i,1]) * C[i,1];
```

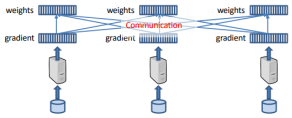
- Reduce the intermediate states which may cause memory references

Transformation of Graphs



Data Parallel Training

- In each iteration:
  - each worker reads different mini-batch of size M, produces gradient
  - communicate to produce aggregated gradient
  - update weights

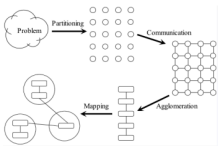


Distributed Training

Distributed training on 1000s cards

- Challenges
  - Communication reduction
    - Minimize the communications between between GPUs to allow high compute efficiency
  - Limited GPU memory
    - Intermediate state for backward computation
  - Load balance
    - Each card should have similar amount of work
  - Fault tolerance
    - MTBF is likely 10-100 hours, while training training usually takes ~1000 hours
    - Quick recovery from system failures

10K – 100K cards for GPT4-like models

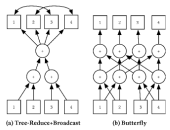


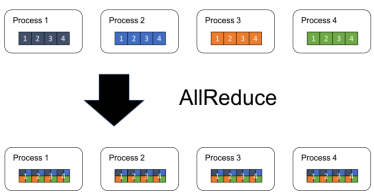
AllReduce to get global gradients

- We need to sum up all the gradients from different machines

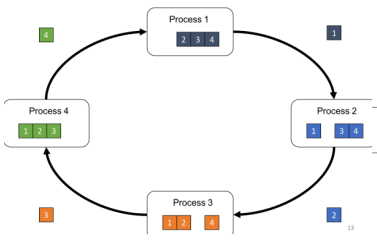
$$G = G_1 + G_2 + \cdots + G_p$$

- Ways to do the summation:

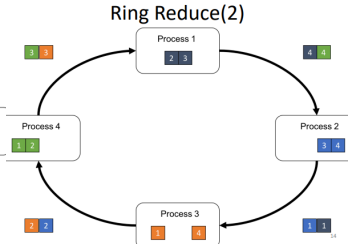




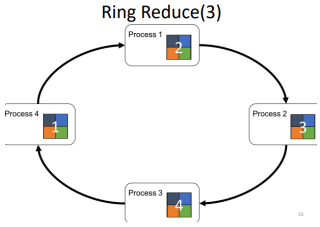
Ring Reduce(1)



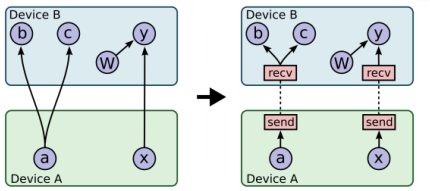
Ring Reduce(2)



Ring Reduce(3)



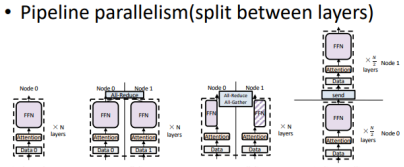
Model Parallel Training



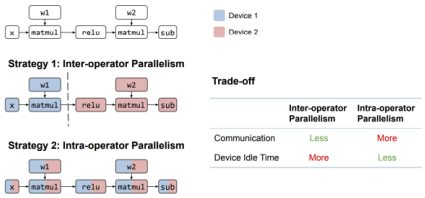

```
import torch.nn as nn

linear1 = nn.Linear(16, 8).to('cuda:0')
linear2 = nn.Linear(8, 4).to('cuda:1')
```

Parallelisms for transformer

- Tensor model parallelism (split a layer)
- Pipeline parallelism(split between layers)

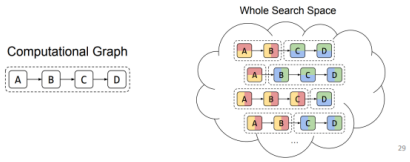


Challenges to computer systems : How to partition the model and choose parallelism?



Can we find the good partition for a large model automatically

- Support both intra and inter-op parallelism, as well as pipeline parallelism
  - There are huge search space and it is a combinatory optimization problem

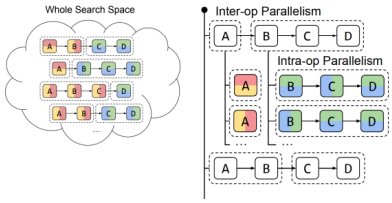


Alpa

- Automatic parallelize the model training with just one line annotation

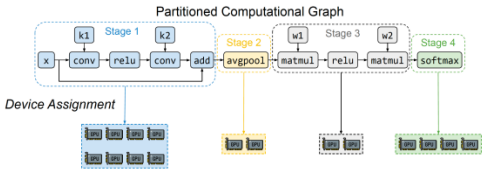
```
@alpa.parallelize
def train_step(model_state, batch):
    def loss_func(params):
        out = model_state.forward(params, batch["x"])
        return np.mean((out - batch["y"]) ** 2)

    grads = grad(loss_func)(state.params)
    new_model_state = model_state.apply_gradient(grads)
    return new_model_state

# A typical JAX training loop
model_state = create_train_state()
for batch in data_loader:
    model_state = train_step(model_state, batch)
```

Two-level Hierarchical space to parallelism

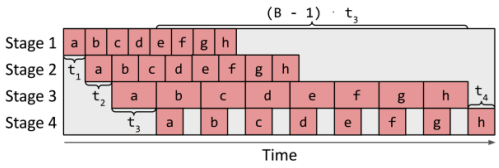


Inter-op parallelism

Graph partition and device assignment within one compiler stage



Inter-op parallelism

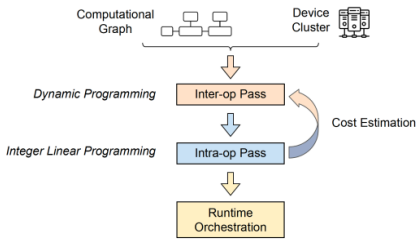


Time = (t1 + t2 + t3 + t4) + (B-1) t3

- Generalized objective function

$$T^* = \min_{\substack{s_1,\ldots,s_S;\\ (n_1,m_1),\ldots,(n_c,m_c)}} \left\{ \sum_{i=1}^{S} t_i + (B-1) \cdot \max_{1 \le j \le S} \{t_j\} \right\}$$

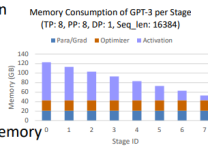
Alpa Compiler phase



Memory Consumption

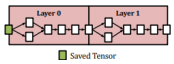
More on memory and load balance

- Large memory consumption
  - Optimizer
  - Parameters/Gradients
  - Activations
- Activations occupy huge memory
- Imbalanced memory consumption
  - Recompute to save memory



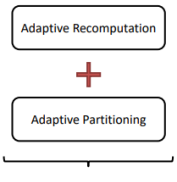
Recomputation

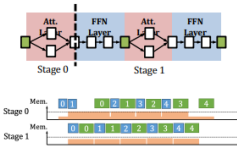
- Drop activations in the forward pass
- Recompute the activations in the backward pass
- Increased computation overhead



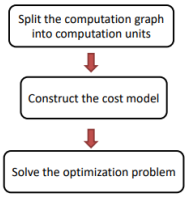
AdaPipe

AdaPipe



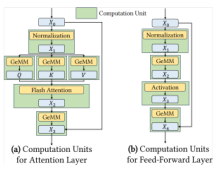


Adaptive Recomputation



Computation Units

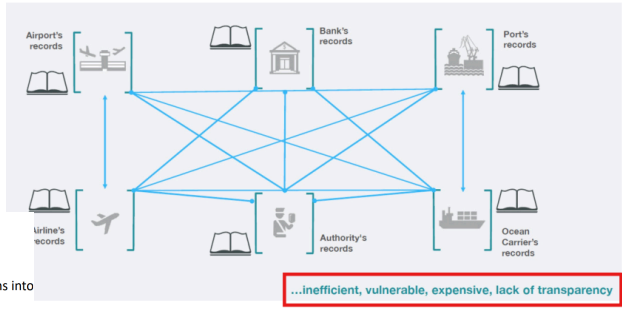
- Fine-grained partitioning
- Operators that require saving output tensors becomes the boundaries.
- Each Tensor bounds to one unit.



Blockchain

Motivation: Why correspondent banks are necessary– Lack of trust

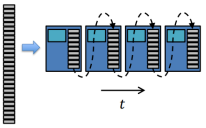
Requires interfaces between each pair of entities



Challenges of a global ledger

- Who can join this ledger?
  - Permissioned or permissionless
- Performance and availability
  - Global network latency and network partition
- Who can write what on this ledger?
  - How to avoid fake transactions?
  - How to avoid double spend?
  - How to avoid modification to previous transactions?
- Who should provide computers to maintain this ledger?
  - Why are they interested in doing this - Incentive

Performance and availability



- Batching transactions into blocks
  - Every 10 minutes 1MB block
- Large number of replicas
  - Every (full) node in the system has a copy

Who can join the ledger?

- Permissionless
  - Public chain
- Permissioned
  - Within a single organization
    - Private chain
  - Multiple party
    - Consortium Blockchain

HYPERLEDGER

RSA (Rivest–Shamir–Adleman)

It is commonly utilized to ensure secure communication and for creating digital signatures. It Uses large integer prime numbers for key generation. It Encrypts data with the public key and decrypts with the private key. It is Slower than some other algorithms but offers strong security.

Key Generation

- Choose/Select two large prime numbers, p and q.
- Calculate $n = p * q$.
- Calculate ϕ(n) = (p-1)(q-1), where ϕ is Euler's totient function.
- Choose an integer e, that $1 < e < \varphi(n)$ and $gcd(e, \varphi(n)) = 1$.
- Compute d, the modular multiplicative inverse of e modulo ϕ(n).
- Public key: (e, n)
- Private key: (d, n)

Encryption

- Convert plaintext message into an integer m.
- Compute ciphertext $c = m^e mod n$.

Decryption

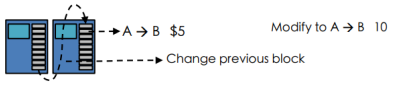
- Calculate the plaintext message $m = c^d mod n$.

Select two prime no's. Suppose **P = 53 and Q = 59.**
Now First part of the Public key : n = P*Q = 3127.
We also need a small exponent say **e :**
But e Must be an integer. Not be a factor of Φ(n).
1 < e < Φ(n) [Φ(n) = (P-1)(Q-1) = 3016],
Choose e =3.public key (e,n) = (3, 3127)

d = (k*Φ(n) + 1) / e for some integer k
For k = 2, value of d is 2011.
Private key (d,n) = (2011, 3127)

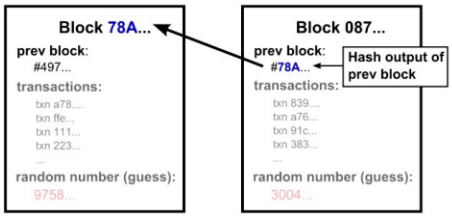
Encrypted Data c = (89^e)mod n = 1394
Decrypted Data = (c^d)mod n = 89

Immutable data

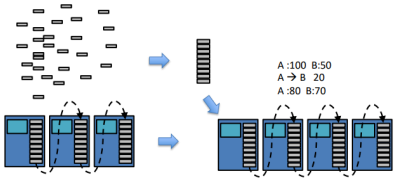


- Crypto-hashing
  - Sha256
  - Single way function,sha256(byte array) = 256bits hash
  - Hard to construct byte array with a given hash
  - A block id which is the hash of the transactions in the block
  - A block contains the blockid of the previous block

Hash outputs = Block IDs

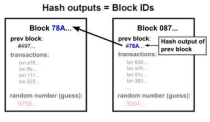


Consensus : Agree on the content of transactions in the block



The Longest chain rule

Proof of work

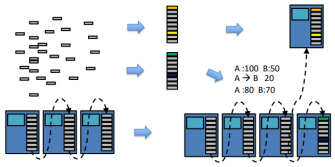


- It is hard to find the content to generate a small hash, but easy to check
- Mining
- Hash generating
  - CPU→GPU→FPGA→ASIC
- Challenge becomes more and more difficult

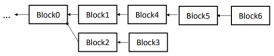
Choose transactions that:

Sha256(prevID，transactions，guess) < 00...00010000

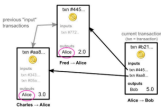
What happens if two of the people solve the challenges at (almost) same time ? -- fork



How to know if a transaction is Unspent?



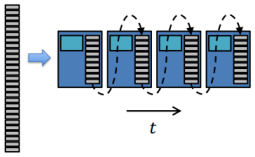
- Track the whole block chain?
  - 873,585 today
  - A large data structure contains many spent output
- A UTXO database
  - Only unspent transactions



Bitcoin blockchain

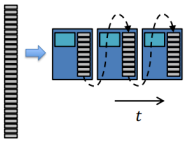
- Immutable, all transactions are pu

Put it together : The bitcoin blockchain



- A system which integrates cryptography, distributed systems, game theory, monetary finance
- Code is law
- Infrastructure for trust

(4) Why people are volunteering to maintain the blockchain systems



- Consortium chain
  - Not a problem
- Public chain
  - Incentive system
  - Reward to generate a block
    - » 50→25→12.5→6.25
  - Transaction fee for each Tx （0.5% today)

Challenges with block chains

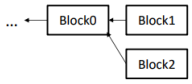
- Public chain
  - Poor scalability
  - Slow（Bitcoin：7 TPS , Taobao，VISA： 100K TPS(peak) , UBER：100TPS）
- Consortium chain
  - Poor implementation
- Security
  - Both on systems and smart contract
- Privacy
  - Zero knowledge proof and Zcash
  - Content

Problems of block chain

- Mining
  - Expensive(ASIC, GPU, CPU）
  - Energy inefficient

Forks in Nakamoto Consensus

Scalability:
What if we run Nakamoto Consensus with **large/fast** blocks generation?

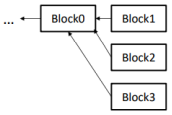


Large blocks → Longer propagation time

Fast block generation → Less time for propagation

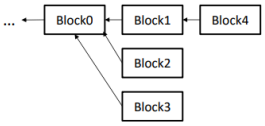
A new block is generated before the network propagates the previous block sufficiently

Forks in Nakamoto Consensus



More forks when the block generation is fast and

Forks in Nakamoto Consensus



Usually the system eventually converges on one fork by waiting for more blocks appended in the end

Solution: **Projects:**

Centralize

7 nodes 21 nodes 1 node

Wait! Why we do blockchain in the first place?

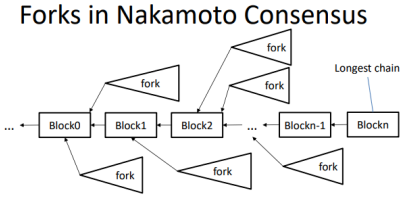
Off-chain

Great idea. But need large players to deposit on the chain.

Sharding

Against CAP Theorem -- rely on economical mechanism to secure each shard.

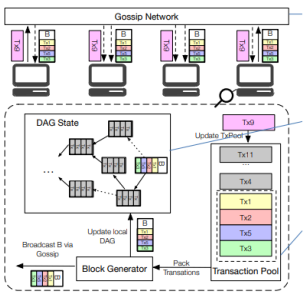
Forks in Nakamoto Consensus



- Forks waste network/processing resources
- Downgrade safety
  - Attacker needs less resource to beat the longest chain

Conflux

Conflux Architecture



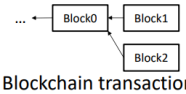
Propagate transactions and blocks

Run consensus on the DAG

Maintain a pool of pending transactions

One Observation

- Bitcoin enforces a restrictive transaction total order at the generation time of each block:

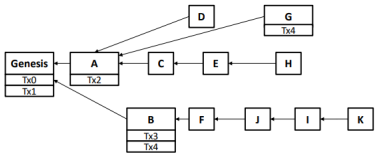


- Blockchain transactions rarely conflict with each other and they can be serialized in any order
- Why not process non-conflicting transactions in **all** concurrent blocks?

Conflux Consensus Ideas

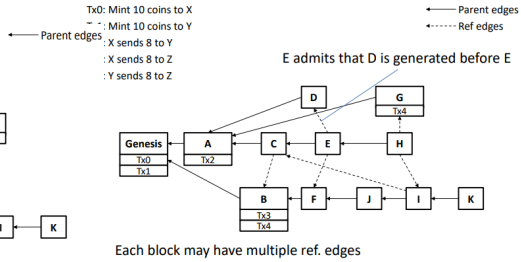
- **Optimistically** process concurrent blocks without discarding any as forks
- Organize blocks into a **direct acyclic graphs (DAG)**
- Enable **fast/large block generation** without sacrificing security
- First agree on a total order of all blocks
  - Assume transactions would not conflict each other
- Then derives the transaction order from the agreed block order
  - Resolve transaction conflicts **lazily**

Tx0: Mint 10 coins to X
Tx1: Mint 10 coins to Y
Tx2: X sends 8 to Y
Tx3: X sends 8 to Z
Tx4: Y sends 8 to Z



Each block has one outgoing parent edge to its parent block

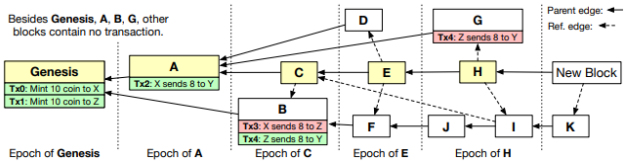
Parent edges form a tree



Each block may have multiple ref. edges

Ref. edges simply denote happens-before relationships

We still need a total order, how?



Computer Systems

Why is it difficult to build a system?

Scales of Computer Systems

- Programming / Data Structure
  - LoC (Lines of Code): From hundreds to millions
- Operating System / Architecture
  - Cores: from one to hundreds
- Network
  - Nodes: from two to millions
- Web Service
  - Clients: from tens to millions

Complexity of Computer Systems

- Hard to define; symptoms:
  - Large number of components
  - Large number of connections
  - Irregular
  - No short description
  - Many people required to design/maintain
- Technology rarely the limit
  - Indeed tech opportunity is the problem
  - Limit is usually designers' understanding

Common Problems of Systems

Problem Types

- **Emergent properties** (surprise!)
  - The properties that are not considered at design time
- **Propagation of effects**
  - Small change -> big effect
- **Incommensurate scaling**
  - Design for small model may not scale
- **Trade-offs**
  - Waterbed effect

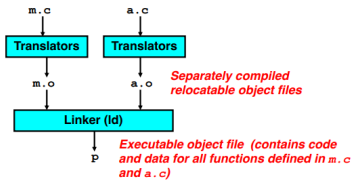
- Waterbed Effect
  - Pushing down on a problem at one point
  - Causes another problem to pop up somewhere else

Coping with Complexity

M.A.L.H

- **Modularity**
  - Split up system
  - Consider separately
- **Abstraction**
  - Interface/Hiding
  - Avoid propagation of effects
- Layering
  - Gradually build up capabilities
- Hierarchy
  - Reduce connections
  - Divide-and-conquer

Use linker to support modularity in software design



How pointers break modular boundaries

```
int buf[2] = {1, 2};
int main()
{
  swap();
  return 0;
}
```
main.c

```
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```
Global

swap.c

Layering

- Goal
  - Reduce module interconnections even more
- How to do it
  - Build a set of mechanisms first(a lower layer)
  - Use them to create a different complete set of mechanisms (an upper layer)
- General rule: A module in one layer only interacts with:
  - Its peers in the same layer, and
  - Modules in the next lower layer / next higher layer

Unbounded Composition

- Two properties of computer systems
  - 1. Mostly digital
  - 2. Controlled by software
  - Both relax the limits on complexity arising from physical laws in other systems

Prep. Exam

1. (15 points) In a storm application, a Spout with <sid1> sends a tuple to Bolt1 whose stid is 0010. It's acker is Ack1. After the execution of Bolt1, it sends two tuples to Bolt2 and Bolt3 respectively.

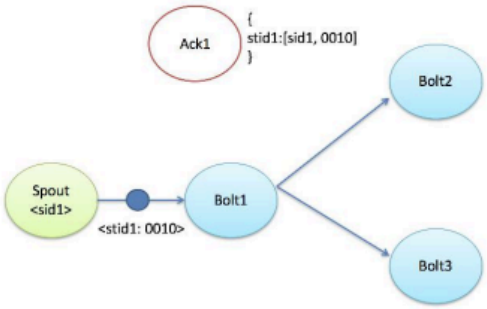
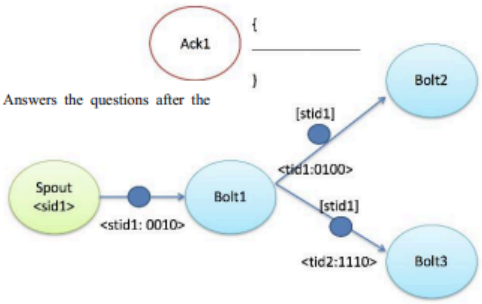


Figure 1.1 Spout sid1 sends a tuple to Bolt 1



essages to Bolt2 and Bolt3

3. (15points) For the following MPI algorithm. Answers the questions after the source code.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "mpi.h"
int main ( int argc, char *argv[] );
int prime_number ( int n, int id, int p );

int main ( int argc, char *argv[] )
{
  int i, id, ierr, n, n_factor, n_hi, n_lo, p, primes, primes_part;
  double wtime;
  n_lo = 1;
  n_hi = 262144;
  n_factor = 2;
  ierr = MPI_Init ( &argc, &argv );
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &p );
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &id );

  for (n = n_lo; n <= n_hi; n *= n_factor) {
    if ( id == 0 )
      wtime = MPI_Wtime ( );
    primes_part = prime_number ( n, id, p );
    ierr = MPI_Reduce ( &primes_part, &primes, 1, MPI_INT ,
__MPI_SUM__ , 0, MPI_COMM_WORLD );

    if ( id == 0 ){
      wtime = MPI_Wtime ( ) - wtime;
      printf ( "  %8d  %8d  %14f\n", n, primes, wtime );
    }
  }
  ierr = MPI_Finalize ( );
  return 0;
}

int prime_number ( int n, int id, int p )
/*
  Purpose:
    PRIME_NUMBER returns the number of primes between 1 and N.
  Parameters:
    Input, int N, the maximum number to check.
    Input, int ID, the ID of this process,
    between 0 and P-1.
    Input, int P, the number of processes.
    Output, int PRIME_NUMBER, the number of prime numbers up to N.
*/
{
  int i, total, prime;
  total = 0;

  for ( i = 2 + id; i <= n; i = i + p ){
    prime = 1;
    for ( j = 2; j < i; j++ )
      if ( ( i % j ) == 0 )
      {
        prime = 0;
        break;
      }
    total = total + prime;
  }
  return total;
```

2. (15 points) In graph computing frameworks, sometimes it is useful to sort vertex with the degree, such as the bottom-up algorithm for BFS. Now we need to write a spark program for this task.

Input:

An unsorted edge list of a directed graph with format(source_vertex_id, destination_vertex_id), the vertex id is a unsigned integer less than 4 billion. The edge list is in text format, one edge per line. For example.

1,2
2,4
3,5
1,4

Output:

Vertex id sorted by in-degree with descending order. If two vertices have the same degree, they are sorted by vertex id with ascending order. The output of the above example is:

4
2
5
1
3

Question:

Write a spark program to perform the task.

```
lines = sc.textFile(input)
edges = lines.map(lambda line: tuple(line.split(',')))
degree_v = edges.map(lambda e: (e[1], 1)).reduce(add).sortBy(lambda x: (-x[1], x[0])).collect()
for _d, v in degree_v:
    print(v)
```

a) Please fill the blanks in the Figure 1.2 to show the record of stid1. The format example is shown in Figure 1.1.
After Bolt1 send ack to Ack1, record will be {stid1: [stid1: 1010]}

b) How would the system detect that a message is not processed completely and need to be reprocessed?
If a bolt crashes, or emits a failure not ack, or the tuple reaches timeout.

c) What is the feature of the Spout to enable the feature of message reprocessing? What is the typical software used to support this feature?
Replayable source. Queue (Kafka).

Question 1: Fill in the blanks in the first page.

Question 2: Is this a load balanced algorithm? How many numbers are processed by each process with number n and process number p(roughly) in one iteration?
Yes. (n-2)/p.

Question3: Any ideas to optimize this code?
Many ways. For example, for (j = 2; j < i; j++) -> for (j = 2; j*j < i; j++).